Le Programmeur

Le langage C



Apprenez rapidement et simplement les bases du langage C

Peter Aitken
Bradley L. Jones

Édition revue et complétée par **Yves Mettier**



LE PROGRAMMEUR

Le langage C

Peter Aitken
et Bradley L. Jones

Édition revue et complétée par Yves Mettier



Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Mise en pages : TyPAO

ISBN : 978-2-7440-4085-6
Copyright© 2009 Pearson Education France
Tous droits réservés

Titre original : *Teach Yourself C in 21 Days, Fourth Edition*

Traduit de l'américain par :
Christine Eberhardt, Emmanuel Simonin
et Jérôme Duclos

Nouvelle édition française revue, corrigée
et complétée par Yves Mettier

ISBN original : 0-672-31069-4
Copyright © 1997 Sams Publishing
All rights reserved.

Sams Publishing
800 East 96th Street
Indianapolis, Indiana 46290 USA

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2˚ et 3˚ a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

Sommaire

Table des matières

Préface à l'édition française 2008

Le langage C est un langage ancien qui date des années 1970 et est toujours d'actualité. C'est un langage relativement simple à apprendre et à mettre en œuvre et un langage puissant, si puissant que, quarante ans après sa création, il reste la référence en matière de programmation.

Cet ouvrage, que nous vous remercions d'avoir acheté, vous présente le C en vingt et un chapitres, ce qui devrait vous permettre, comme l'indique le titre original *Teaching yourself in 21 days*, d'apprendre ce langage en trois semaines à raison de un chapitre par jour. À la fin de ce livre, vous serez apte à réaliser de petits programmes et à comprendre le code des plus gros. Mais, à l'aide de bibliothèques de fonctions existantes, vous pourrez créer vos interfaces graphiques, communiquer avec d'autres programmes sur Internet, réaliser des jeux ou traiter des données issues des bases de données.

Ce que cette nouvelle édition apporte

Cette nouvelle édition a pour origine la traduction en français de *Teaching yourself in 21 days*, qui date de 1995 pour la version originale et de 1997 pour la traduction. Souvenez-vous, en 1995, Microsoft publiait le légendaire système Windows 95. Mais MS-DOS occupait encore bon nombre d'ordinateurs. La société Apple était au contraire en grande difficulté (au point de changer de PDG début 1996). Le monde Unix était réservé aux professionnels. Quant à Linux, il n'avait que quatre ans et n'intéressait que les amateurs, dont un grand nombre d'étudiants. En 1995, les programmeurs en C suivaient encore pour beaucoup la norme ANSI alors que la norme ISO C89 était parue six ans auparavant. Mais elle n'était pas suffisamment supportée pour être considérée comme le nouveau standard du C.

Les ordinateurs étaient encore sur 16 bits et les nouveaux Pentium sur 32 bits venaient d'apparaître.

En 2008, les choses ont changé. MS-DOS a disparu et les versions de Microsoft Windows se sont succédé, gagnant en stabilité. Mais, surtout, Unix est revenu en force avec le succès inattendu de GNU/Linux, la nouvelle version de Mac OS X, dont la base, appelée Darwin, n'est rien d'autre qu'un Unix, et dans le milieu professionnel l'amélioration des systèmes Unix existants. Le succès des logiciels libres a permis à tous de disposer de systèmes d'exploitation libres et gratuits et, pour les programmeurs, de développer de plus en plus de fonctionnalités. Les ordinateurs 32 bits sont de rigueur et le 64 bits commence à apparaître chez les particuliers.

En une dizaine d'années, le C semble être le seul à ne pas avoir bougé et l'on pourrait se prendre à penser que c'est un témoin des années 1970. Mais, détrompez-vous, ce qui devait être une simple relecture et mise à jour de cet ouvrage s'est révélé un véritable travail d'adaptation. Le C a peu changé, mais les ordinateurs ont évolué, les hommes aussi. Voici ce que cette nouvelle édition apporte.

Linux et MS-DOS, 64, 32 et 16 bits

La plupart des références à MS-DOS ont été remplacées et adaptées à Linux. Tous deux ont cette similarité de disposer d'une interface en ligne de commande. À de rares exceptions près, ce qui était valable pour MS-DOS l'est pour Linux dans ce livre. En revanche, le passage de 16 bits à 32 bits a été plus délicat. Cela a concerné d'une part les entiers de type `int` et d'autre part les pointeurs. Différencier un `int` qui faisait autrefois 2 octets et un `float` de 4 octets devient sans intérêt. Il a fallu adapter les exemples soit en remplaçant les `int` en `short`, qui ont toujours une taille de 2 octets, soit en remplaçant les `float` par des `double`. Quant aux pointeurs, ils nous ont obligé à refaire de nombreuses figures.

Considérations de sécurité

Les aspects de sécurité informatique sont apparus dans les années 1990 avec Internet, qui a démultiplié le nombre de virus et autres vers informatiques. Internet a également permis aux pirates informatiques de pénétrer plus facilement les ordinateurs en s'y connectant à distance et en profitant de failles des programmes. Dans ce livre, la plus flagrante était l'utilisation massive de la fonction `gets()`, qui permet de lire une ligne au clavier. Très simple d'emploi, elle est très prisée des programmeurs débutants. Cependant, cette instruction est à elle seule une faille de sécurité. En effet, elle n'effectue aucun contrôle sur la longueur de la chaîne de caractères que l'utilisateur lui donne. S'il est mal intentionné, il peut envoyer plus de caractères que le programme ne peut en accepter, ce qui peut entraîner un plantage de celui-ci, voire pire. Vous devez retenir deux choses de cela :

- N'utilisez JAMAIS la fonction `gets()`. Vous pouvez utiliser `fgets()` à la place.

- `fgets()` n'étant pas tout à fait équivalente à `gets()`, et pour ne pas avoir à réécrire tous les exemples du livre, nous avons écrit une autre fonction, `lire clavier()`, quasi équivalente à `gets()`. Vous trouverez son code dans l'exemple pratique 1. Vous devrez recopier la définition de cette fonction dans tous les exemples qui y font appel.

Dans le même ordre d'esprit, la fonction `scanf()` peut, mal employée, faire apparaître la même faille de sécurité que `gets()`. Si vous utilisez `scanf()`, ne mettez jamais `"%s"` dans sa chaîne de format. Indiquez toujours une longueur maximale de chaîne, comme `"%10s"` pour un maximum de 10 caractères.

Pour résumer cette section très importante, N'UTILISEZ JAMAIS `gets()` ET NE METTEZ JAMAIS `"%s"` DANS LE FORMAT DE `scanf()`.

Conventions de codage

Une dernière modification générique du code a consisté à faire terminer tous les programmes par un appel à `exit(EXIT SUCCESS)` ou `exit(EXIT FAILURE)` selon le cas. Cela implique l'inclusion du fichier d'en-têtes *stdlib.h* qui a été ajouté au début des exemples lorsqu'il n'y était pas déjà. Nous nous trouvons ici dans les conventions de codage car vous trouverez souvent `return 0` ou `exit(0)` à la place de `exit(EXIT SUCCESS)` et la différence est faible. Nous recommandons en fait l'utilisation des constantes prédéfinies lorsqu'elles existent et que cela est possible.

Il existe d'autres conventions de codage que nous n'avons pas souhaité appliquer ici car ce sont plus des habitudes (des bonnes) que des règles de programmation. Ainsi, si vous écrivez un test de comparaison entre une variable et un nombre, par exemple `x == 3`, il est préférable d'écrire `3 == x`. En effet, dans ce cas, si vous oubliez un signe égal, la première expression attribue la valeur 3 à la variable `x` alors que la seconde est tout simplement invalide (il est impossible d'affecter `x` à `3`). Dans le premier cas, vous introduisez un bogue alors que, dans le second cas, le compilateur verra l'erreur et ne manquera pas de vous la signaler. Pour d'autres conventions de codage, nous vous recommandons la lecture des *GNU Coding Standards*, disponible à l'adresse **http://www.gnu.org/prep/standards/**.

Gestion de la mémoire : faites ce que je dis, pas ce que je fais

Une modification importante qui n'a pas été faite dans ce livre concerne les allocations de mémoire (avec `malloc()`, par exemple). Celle-ci est en effet rarement libérée dans les exemples (avec `free()`). Nous avons hésité à reprendre les exemples mais leur lisibilité l'a emporté sur la rigueur pour faciliter la compréhension. Dans l'analyse, il est rappelé que vous devez libérer la mémoire. En d'autres termes, *faites ce que je dis, pas ce que je fais*. Dans vos programmes, cela est essentiel car, si la mémoire n'est plus

limitée à 640 kilo-octets comme au temps de MS-DOS, elle n'est toujours pas extensible et, surtout, elle est maintenant partagée entre les diverses applications qui tournent en parallèle à votre programme sur votre ordinateur.

La programmation en C aujourd'hui

Le langage C a peu évolué. Depuis les années 1970 sont sorties les normes C89, largement supportée par les compilateurs, et C99, plus récente. Mais, depuis dix ans, le monde a changé. D'autres normes sont sorties. D'autres besoins sont apparus. Des outils et des bibliothèques de fonctions ont été écrits.

Les normes

Les normes C89 et C99 sont assez proches de la norme ANSI d'origine. Ces normes cadrent le langage C. Mais il fallait également normer les systèmes d'exploitation. En effet, cela peut sembler agréable d'entendre dire que le langage C est très portable et existe sur la plupart des plates-formes. Mais à quoi bon cette portabilité si vos programmes fonctionnent de façon différente sur chacune d'elles. Pour cela, une autre norme a été créée pour les systèmes d'exploitation. Il s'agit de la norme POSIX. La plupart des systèmes d'exploitation grand public respectent cette norme. C'est le cas entre autres de Windows NT et Vista (à condition d'activer certaines fonctionnalités optionnelles), de GNU/Linux et de Mac OS X. Il existe également d'autres normes comme BSD (BSD4.3 ou BSD4.4, par exemple), SVr4 (System V Release 4). Ces normes sont également répandues et vous pouvez vous y fier.

Les besoins

Les besoins en termes de programmation ont évolué par rapport à il y a quelques années. Avec l'essor des logiciels libres et l'augmentation du nombre de fonctionnalités inhérentes aux systèmes d'exploitation, les nombreux petits utilitaires que les programmeurs développaient pendant leur temps libre sont intégrés et n'ont plus besoin d'être écrits. Par exemple, rares sont ceux qui vont écrire un énième explorateur de fichiers.

Les technologies évoluent aussi. Par exemple, il était autrefois simple d'imprimer sur une imprimante matricielle branchée au port parallèle de votre ordinateur. Aujourd'hui, l'imprimante est reliée au port USB quand elle n'est pas connectée à un serveur d'impression. De plus, l'amélioration de nos écrans et l'augmentation de leur taille ne nécessitent plus forcément d'imprimer autant. Nous lisons de plus en plus nos documents en ligne.

Ce livre a dû évoluer avec nos besoins. Ainsi, l'impression des documents a été supprimée car, pour imprimer un simple texte, vous utiliserez les fonctionnalités de redirection de la

ligne de commande. Dans les cas plus complexes où il s'agit de mettre en page du texte et des images, nous sortons du cadre de ce livre.

Les outils

Dans les années 1970 et peut-être encore un peu dans les années 1990, la programmation en C était assez limitée par la norme ANSI. Vous deviez ensuite utiliser des bibliothèques de fonctions généralement commerciales et payantes pour utiliser leurs fonctionnalités et arriver à vos fins. L'essor des logiciels libres, à nouveau, a permis aux développeurs d'isoler leurs fonctions génériques dans des bibliothèques de fonctions et de les diffuser pour une utilisation libre. Alors qu'autrefois il était à peine pensable d'embarquer une fonctionnalité de compression de données dans votre programme, cela est aujourd'hui tout à fait naturel, à l'aide de la bibliothèque adéquate (par exemple *zlib*), de compresser en quelques lignes de code. De la même façon, les cours et livres d'algorithmie présentaient un grand intérêt pour organiser les données dans des structures optimisées pour leur traitement. Aujourd'hui, des bibliothèques dédiées démocratisent certains algorithmes, même évolués. Par exemple, utiliser une table de hachage nécessite quelques lignes de code avec la bibliothèque *glib* alors que cela se comptait en centaines de lignes lorsqu'il fallait tout faire. Bien que tout cela sorte du cadre de ce livre, nous vous présentons en annexe quelques bibliothèques qui vous seront bien utiles pour continuer à programmer en C au-delà de ce que ce livre vous aura appris.

Un des outils les plus importants si ce n'est le plus important en programmation en C est le compilateur. En l'occurrence, le projet GNU a développé le sien, GCC (initialement *GNU C Compiler* et aujourd'hui renommé en *GNU Compilers Collection*). Ce compilateur, qui est celui par défaut sur Linux, a été porté sur de nombreuses architectures dont Windows et Mac OS X, parfois tel quel, parfois en s'intégrant à des suites logicielles comme la suite XCode chez Apple ou l'environnement de développement intégré (EDI) WxDev-C++ Grâce à lui, vous disposez d'un compilateur libre et gratuit sur denombreuses plates-formes. Nous avons donc dû faire évoluer ce livre pour le prendre encompte comme, *a priori*, votre compilateur alors que, dans l'édition originale, il s'agissait de Turbo C (Borland) ou de Visual C++ (Microsoft).

Enfin, dans certains cas, il existe même des outils pour nous faciliter certaines tâches. Nous citerons par exemple les autotools (autoconf, automake) pour automatiser la compilation et la distribution de vos programmes en simplifiant parfois des problèmes de portabilité qui pourraient se poser. Nous citerons également les outils d'internationalisation comme gettext, qui vous génère du code et des fichiers pour simplifier la tâche de traduction, aussi bien au programmeur, qui aura peu de travail pour permettre la traduction de son programme, qu'au traducteur, qui n'aura plus besoin de compétences poussées en programmation pour traduire.

La programmation système et réseau

Deux des domaines de prédilection du langage C sont la programmation système et réseau. À un haut niveau, il est préférable d'utiliser des bibliothèques qui vous facilitent la tâche. Vous trouverez le nom de certaines de ces bibliothèques en annexe. Cependant, à un niveau plus bas, le langage C et la bibliothèque standard libc fournissent un jeu de fonctions assez important. Vous pouvez par exemple paralléliser l'exécution de certaines parties de votre code. Vous pouvez également créer un programme résident (autrement appelé démon), un serveur ou un client pour se connecter au serveur, réagir à des signaux (comme l'appui sur les touches Ctrl+C), partager de la mémoire entre plusieurs programmes...

Ce sujet est un sujet à part entière et nous, auteurs et relecteur, n'avons pas souhaité le traiter de manière approfondie dans cet ouvrage et encore moins le survoler. Si ce domaine vous intéresse, nous vous conseillons de chercher sur Internet un des nombreux tutoriels ou sites de documentation sur ce sujet ou d'acquérir un des quelques livres en français qui en parlent. Nous citerons en particulier *Programmation Linux en pratique* (CampusPress, 2007) d'Arnold Robbins et *C en action* (O'Reilly, 2005) d'Yves Mettier, relecteur de cet ouvrage.

Remerciements

Ayant effectué un grand travail de relecture et de mise à jour de cet ouvrage, je voudrais remercier Patricia Moncorgé (Pearson Education) de m'avoir proposé de le faire, ainsi que ma famille pour m'avoir soutenu, en particulier Anabella et notre petit Matthieu.

Yves Mettier

Auteur du *Livre de recettes C en action* (O'Reilly, 2005)
et du *Guide de survie Langage C* (Pearson, 2007)

Introduction

Ce livre a été conçu pour que vous maîtrisiez le langage C à la fin des vingt et un chapitres. Malgré la concurrence de langages plus récents tels que Java ou C++, le langage C reste un bon choix pour débuter en programmation. Vous découvrirez pourquoi vous avez eu raison de le choisir au Chapitre 1.

Cet ouvrage présente le langage C de la manière le plus logique possible. Votre progression en sera d'autant plus facilitée. Nous avons conçu ce livre pour vous permettre d'aborder les chapitres à raison de un par jour. Nous avons supposé que vous n'avez aucune expérience de la programmation. Bien sûr, si vous avez déjà utilisé un autre langage, comme le basic, vous apprendrez plus vite. Uniquement consacré à l'étude du langage C, ce manuel s'applique à tout type d'ordinateur ou de compilateur.

Caractéristiques de ce livre

Ce manuel a adopté un certain nombre de conventions pour vous aider à reconnaître des types d'informations spécifiques. Les paragraphes "Syntaxe" apportent tous les détails nécessaires à l'utilisation d'une commande ou d'un concept particulier. Leurs explications sont illustrées par des exemples. Les lignes qui suivent donnent un aperçu de ce type de paragraphe. (Nous aborderons ces notions dès le Chapitre 1 !)

Syntaxe de la fonction *printf()*

```
#include <stdio.h>
printf( chaîne-format [, arguments,...]);
```

`printf()` est une fonction qui peut recevoir des *arguments*. Ceux-ci doivent correspondre en nombre et en type aux spécifications de conversion contenues dans la chaîne format. `printf()` envoie les informations mises en forme vers la sortie standard (l'écran). Pour qu'un programme puisse appeler cette fonction, le fichier standard d'entrées/sorties stdio.h doit avoir été inclus.

La *chaîne-format* est requise mais les arguments sont facultatifs. Elle peut contenir des ordres de contrôle. Voici quelques exemples d'appels de la fonction `printf()` :

Exemple 1 : code

```
#include <stdio.h>
int main()
{
printf("voici un exemple de message !");
}
```

Exemple 1 : résultat

```
voici un exemple de message !
```

Exemple 2 : code

```
printf("ceci affiche un caractère, %c\nun nombre, %d\nun nombre virgule \
        flottante, %f", 'z', 123, 456.789 );
```

Exemple 2 : résultat

```
ceci affiche un caractère, z
un nombre, 123
un nombre virgule flottante, 456.789
```

Les rubriques "Conseils" sont une autre caractéristique de ce livre. Elles mettent l'accent sur ce que vous pouvez faire et sur ce que vous devrez absolument éviter de faire.



À faire

Lire la fin de ce chapitre. Elle vous donne les détails de la partie située à la fin de chaque chapitre.

À ne pas faire

Sauter les questions du quiz ou les exercices. Si vous pouvez répondre aux questions du contrôle, vous êtes prêt à poursuivre votre étude.

Vous rencontrerez également des rubriques exposant des astuces, des infos, et des mises en garde.



Les astuces vous donnent des raccourcis et des techniques de travail très utiles.



Ces rubriques fournissent des compléments sur le concept C traité.



Ces avertissements signalent les pièges les plus courants.

De nombreux exemples de programmes sont fournis tout au long de ce livre pour illustrer les caractéristiques et les concepts du C. Ces exemples se composent de trois parties : le programme lui-même, les données à lui transmettre et la sortie qu'il génère, puis une analyse ligne par ligne de son fonctionnement.

Un paragraphe Q&R clôt chaque chapitre avec les réponses aux questions les plus courantes. Ce paragraphe est suivi d'un atelier qui propose un quiz et des exercices portant sur les concepts du chapitre. Vous pourrez contrôler la pertinence de vos réponses dans l'Annexe G.

Quoi qu'il en soit, vous ne deviendrez pas un programmeur en langage C simplement en lisant ce livre. Ce n'est qu'en programmant qu'on devient programmeur. Chaque série de questions est suivie d'une batterie d'exercices. Nous vous recommandons de les faire. Créer du code est la meilleure façon de progresser.

Dans les exercices intitulés "**CHERCHEZ L'ERREUR**", vous devrez retrouver les erreurs que nous avons glissées dans le code et les rectifier comme vous aurez à le faire avec vos propres programmes. Si votre recherche est infructueuse, ces réponses sont fournies en Annexe G.

Plus vous avancerez dans ce livre, plus les réponses de certains exercices deviendront longues. D'autres encore peuvent avoir de multiples solutions. C'est pour ces raisons que vous ne retrouverez pas toutes les solutions des derniers chapitres en Annexe G.

Améliorations

Rien n'est parfait, mais on peut approcher de la perfection. Cette édition est la quatrième et nous nous sommes efforcés de vous présenter un code compatible à cent pour cent avec le plus grand nombre possible de compilateurs C. Plusieurs contrôles ont été réalisés pour assurer à ce livre le meilleur niveau technique. Ces contrôles s'ajoutent à ceux des auteurs et aux transformations qui ont suivi les suggestions des lecteurs des trois éditions précédentes.



Le code source présenté dans ce livre a été testé et compilé sur les plates-formes suivantes : DOS, Windows, System 7.x (Macintosh), UNIX et OS/2. Les lecteurs des éditions précédentes ont également utilisé ce code sur toutes les plates-formes supportant le C.

Les sections "Exemple pratique" sont une nouveauté de cette édition. Elles sont au nombre de six et présentent un programme C court qui accomplit une tâche utile ou amusante. L'objectif de ces programmes est d'illustrer des techniques de programmation C. Vous pouvez saisir ces programmes et les exécuter, puis manipuler éventuellement le code pour trouver d'autres applications. Ces sections sont destinées à l'expérimentation. Nous espérons que vous les apprécierez.

Où trouver le code présenté dans ce livre

Vous trouverez le code source des principaux exemples de ce livre sur le site de Pearson Éducation France : **www.pearsoneducation.fr**.

Conventions

Ce livre utilise différentes polices de caractères qui vous aideront à différencier le code C de ses commentaires, et qui mettront en valeur les concepts importants. Le code C est imprimé avec une police de caractères particulière `à largeur fixe`. Les données entrées par l'utilisateur en réponse aux messages des programmes sont représentées **`avec cette même police en caractères gras`**. Les termes qui représentent ce que vous devrez effectivement saisir dans le code C sont imprimés *`en largeur fixe et en italique`*. Les termes nouveaux ou importants sont imprimés en *italique*.

Tour d'horizon de la Partie I

Avant de commencer votre apprentissage du langage C, un compilateur et un éditeur sont nécessaires. Si vous n'avez ni l'un ni l'autre, vous pouvez quand même utiliser ce livre mais la valeur de son enseignement en sera diminuée. La meilleure façon d'apprendre un langage de programmation est de créer et lancer de nombreux programmes. Les exemples donnés dans ce livre offrent un bon support pour les définitions et exercices.

Chaque chapitre se termine par un atelier constitué d'un quiz et de quelques exercices portant sur les sujets étudiés. Les réponses et solutions complètes des premiers chapitres se trouvent dans l'Annexe G. Il n'a pas été possible de prévoir toutes les réponses pour les derniers chapitres car il existe un grand nombre de solutions. Nous vous recommandons de tirer le meilleur parti de ces ateliers et de contrôler vos réponses.

Ce que vous allez apprendre

Cette première partie aborde les notions de base du C. Les Chapitres 1 et 2 vous apprendront à créer un programme C et à en reconnaître les éléments de base. Le Chapitre 3 définit les différents types de variables C. Le Chapitre 4 introduit les instructions et expressions d'un programme pour obtenir de nouvelles valeurs. Il vous explique également comment introduire des conditions dans l'exécution d'un programme avec l'ordre IF. Le Chapitre 5 traite des fonctions du langage C et de la programmation structurée. Le Chapitre 6 concerne les commandes qui permettent de

contrôler le déroulement des programmes. Enfin, le Chapitre 7 vous permettra d'imprimer et de dialoguer avec votre clavier ou votre écran.



Ce livre s'appuie sur le standard C ANSI. Cela signifie que vous pouvez utiliser le compilateur C de votre choix s'il respecte bien la norme ANSI.

1

Comment démarrer

Vous apprendrez dans ce chapitre :

- Pourquoi le langage C représente le meilleur choix d'un langage de programmation
- Les étapes du cycle de développement d'un programme
- Comment écrire, compiler et lancer votre premier programme C
- Comment faire face aux messages d'erreurs générés par le compilateur et l'éditeur de liens

Bref historique du langage C

Le langage C a été créé par Dennis Ritchie aux Bell Telephone Laboratories en 1972. Il a été conçu dans un dessein bien précis : développer le système d'exploitation UNIX, déjà utilisé sur de nombreux ordinateurs. Dès l'origine, il devait donc permettre aux programmeurs de travailler de manière productive et efficace.

En raison de sa puissance et de sa souplesse, l'utilisation du C s'est rapidement répandue au-delà des laboratoires Bell. Les programmeurs ont commencé à l'utiliser pour écrire toutes sortes de programmes. Rapidement, des organisations diverses ont utilisé leurs propres versions du langage C, et de subtiles différences d'implémentation sont devenues un véritable casse-tête pour les programmeurs. En réponse à ce problème, l'American National Standards Institute (ANSI) a formé un comité en 1983 pour établir une définition standard du C, qui est devenu le C standard ANSI. À quelques exceptions près, les compilateurs C d'aujourd'hui adhèrent à ce standard.

Le nom du langage C vient de son prédécesseur qui était appelé B. Le langage B a été développé par Ken Thompson qui travaillait aussi aux laboratoires Bell.

Pourquoi utiliser le langage C ?

Il existe de nombreux langages de programmation de haut niveau comme le C, le Pascal, ou le Basic. Ils sont tous excellents et conviennent pour la plupart des tâches de programmation. Toutefois, les professionnels placent le langage C en tête de liste pour plusieurs raisons :

- Il est souple et puissant. Ce que vous pourrez accomplir avec ce langage n'est limité que par votre imagination. Vous n'aurez aucune contrainte. Le langage C est utilisé pour des projets aussi variés que des systèmes d'exploitation, des traitements de textes, des graphiques, des tableurs ou même des compilateurs pour d'autres langages.
- Lorsqu'une nouvelle architecture (nouveau processeur, nouveau système d'exploitation...) apparaît, le premier langage disponible est généralement le C car contrairement à d'autres, il est facile à porter. De plus, un compilateur C est souvent disponible sur les ordinateurs (à l'exception de Windows malheureusement), ce qui n'est pas le cas pour les autres langages.
- Avec la norme ANSI, le C est devenu un langage portable. Cela signifie qu'un programme C écrit pour un type d'ordinateur (un PC IBM, par exemple) peut être compilé pour tourner sur un autre système (comme un DEC VAX) avec très peu ou

aucune modification. Les règles qui sont à respecter par les compilateurs sont décrites plus loin dans ce livre.

- Le langage C contient peu de mots. Une poignée d'expressions appelées mots clés servent de bases pour l'élaboration des fonctions. On pourrait penser, à tort, qu'un langage possédant plus de mots clés (quelquefois appelés mots réservés) pourrait être plus puissant. Lorsque vous programmerez avec ce langage, vous vous apercevrez que vous pouvez réaliser n'importe quelle tâche.
- Le langage C est modulaire. Son code peut (et devrait) être écrit sous forme de sous-programmes appelés fonctions. Ces fonctions peuvent être réutilisées pour d'autres applications ou programmes. Si vous passez des informations à ces fonctions, vous obtenez du code réutilisable.

Comme vous pouvez le constater, le choix du C en tant que premier langage de programmation est excellent. Vous avez certainement entendu parler de C++. Ce langage s'appuie sur une technique de programmation appelée programmation orientée objet.

C++ était initialement une version améliorée du C, à savoir un C disposant de fonctions supplémentaires pour la programmation orientée objet. Le C++ est aujourd'hui un langage à part entière. Si vous êtes amenés à étudier ce langage, ce que vous aurez appris du C vous aidera grandement.

Un autre langage, également basé sur C, a été l'objet d'une attention toute particulière. Il s'agit de Java. Si vous décidez de vous orienter vers la programmation Java, vous découvrirez rapidement qu'il existe de nombreuses similitudes entre ces deux langages.

Avant de programmer

Vous ne pouvez résoudre que les problèmes que vous aurez identifiés. Il sera alors possible de bâtir un plan pour les corriger. Lorsque vous aurez appliqué ce plan, vous devrez tester les résultats pour savoir si les problèmes ont bien été résolus. Cette logique s'applique à de nombreux domaines, la programmation en fait partie.

Voici les étapes à suivre pour créer un programme en langage C (ou dans n'importe quel autre langage) :

1. Définir les objectifs du programme.
2. Choisir les méthodes que vous voulez utiliser pour écrire ce programme.
3. Créer le programme.
4. Enfin, l'exécuter et observer les résultats.

Un exemple d'objectif (voir étape 1) serait d'écrire un traitement de texte ou un programme de base de données. Un objectif plus simple consiste à afficher votre nom sur l'écran. Si vous n'avez pas de fonction à réaliser, vous n'avez pas besoin d'un programme.

Pour la deuxième étape, vous devez définir vos besoins, la formule à utiliser, et établir un ordre de traitement des informations.

Par exemple, imaginez que quelqu'un vous demande d'écrire un programme pour calculer l'aire d'un cercle. L'étape 1 est réalisée puisque vous connaissez votre objectif : trouver la valeur de cette aire. L'étape 2 consiste à déterminer quelles sont les données à connaître pour le calcul. Si l'utilisateur du programme donne le rayon du cercle, la formule πr^2 vous donnera la réponse. Vous pouvez maintenant passer aux étapes 3 et 4 qui constituent le développement du programme.

Cycle de développement du programme

La première étape du développement d'un programme est la création du code source avec un éditeur. La deuxième étape consiste à compiler ce code pour obtenir un fichier objet. Dans la troisième, vous transformez le code compilé en fichier exécutable. Le lancement du programme dans la quatrième étape permet d'en vérifier les résultats.

Création du code source

Le code source est une série de commandes ou de déclarations qui indiquent à l'ordinateur les tâches que vous voulez lui faire exécuter. C'est la première étape du développement et le code source est créé à l'aide d'un éditeur. Voici un exemple d'instruction de code source C :

```
printf("Bonjour, vous !");
```

Cette instruction demande à l'ordinateur d'afficher le message `"bonjour, vous !"` à l'écran.

Utilisation de l'éditeur

La plupart des compilateurs sont livrés avec un éditeur intégré qui permet de créer le code source. Consultez votre manuel pour savoir si votre compilateur en fait partie.

La plupart des systèmes d'exploitation contiennent un programme qui peut être utilisé comme un éditeur. Si vous travaillez avec UNIX, vous pouvez utiliser vi ou vim, emacs ou un bloc-notes comme gedit ou kedit. Microsoft Windows vous offre le bloc-notes.

Les logiciels de traitement de texte utilisent des codes spéciaux pour formater leurs documents. Ces codes ne peuvent pas être lus correctement par les autres programmes. L'*American Standard Code for Information Interchange* (ASCII) a défini un format de texte standard que n'importe quel programme, y compris le C, peut utiliser. Beaucoup de traitements de texte, comme Open-Office.org, Abiword, Koffice et Microsoft Word, sont capables de sauvegarder des fichiers source en format ASCII (comme un fichier texte plutôt que comme un fichier document). Pour obtenir un fichier en format ASCII avec un traitement de texte, vous devez choisir l'option de sauvegarde ASCII ou texte.

Vous n'êtes pas obligé d'utiliser un de ces éditeurs. Il existe des programmes, que vous pouvez acheter, qui sont spécialement destinés à créer du code source. Citons également les logiciels libres (et gratuits) Ajuta et Kdevelop disponibles au moins sur GNU/Linux.



Pour trouver des éditeurs différents, vous pouvez consulter votre revendeur local, les catalogues de vente par correspondance ou encore les petites annonces des magazines de programmation. Sur votre distribution Linux, effectuez une recherche dans les packages disponibles.

Quand vous sauvegardez un fichier source, il faut lui donner un nom. Vous pouvez choisir n'importe quel nom ou extension, mais il existe une convention : le nom du programme doit représenter la fonction de ce programme et `.C` est reconnue comme l'extension appropriée.

Compilation du code source

Votre ordinateur ne peut pas comprendre le code source C. Il ne peut comprendre que des instructions binaires dans ce que l'on appelle du langage machine. Votre programme C doit être transformé en langage machine pour pouvoir être exécuté sur votre ordinateur. Cela représente la deuxième étape de développement du programme. Cette opération est réalisée par un compilateur qui transforme votre fichier code source en un fichier contenant les mêmes instructions en langage machine. Ce fichier créé par le compilateur contient le code objet , et on l'appelle fichier objet.



Ce livre s'appuie sur le standard C ANSI. Cela signifie que vous pouvez utiliser le compilateur C de votre choix s'il respecte bien la norme ANSI.

Chaque compilateur possède sa propre commande pour créer du code objet. En général, il faut taper la commande de lancement du compilateur suivie du nom du fichier source. Voici quelques exemples de commandes destinées à compiler le fichier source `radius.c` en utilisant divers compilateurs DOS/Windows :

Compilateur	***Commande***
Gnu gcc	`gcc radius.c`
C Microsoft	`cl radius.c`
Turbo C de Borland	`tcc radius.c`
C Borland	`bcc radius.c`
Compilateurs C Unix	`cc radius.c`

La compilation sera simplifiée dans un environnement de développement graphique. Dans la plupart des cas, cette opération sera réalisée à partir du menu ou de l'icône correspondante. Une fois le code compilé, il suffira alors de sélectionner l'icône en cours ou la touche du menu adéquate pour exécuter le programme. Pour de plus amples renseignements vous vous référerez au manuel de votre compilateur.

Après cette opération, vous trouverez dans votre répertoire courant un nouveau fichier ayant le même nom que votre fichier source, mais avec l'extension `.o` ou `.obj`. Cette extension sera reconnue par l'éditeur de liens comme celle d'un fichier objet.

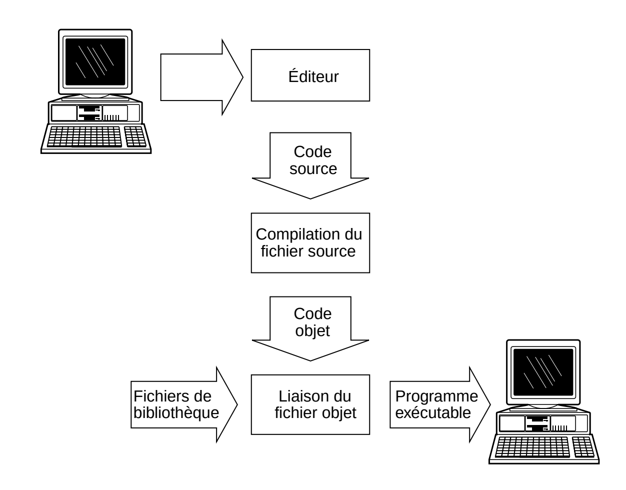
Création du fichier exécutable

Une partie du langage C est constituée d'une *bibliothèque de fonctions* contenant du code objet (ce code a déjà été compilé) destiné à des *fonctions prédéfinies*. Ces fonctions sont fournies avec votre compilateur et `printf()` en est un exemple.

Ces fonctions réalisent des tâches très souvent réalisées comme afficher des informations à l'écran ou lire un fichier. Si votre programme les utilise, le fichier objet obtenu après compilation doit être complété par le code objet issu de la bibliothèque de fonctions. Cette dernière étape, appelée liaison, fournit le programme exécutable (*exécutable* signifie que ce programme peut être exécuté sur votre ordinateur).

La Figure 1.1 représente le schéma de la transformation du code source en programme exécutable.

Figure 1.1
Le code source est transformé en code objet par le compilateur puis en fichier exécutable par l'éditeur de liens.



Fin du cycle de développement

Une fois que vous avez obtenu votre fichier exécutable, vous pouvez lancer votre programme en saisissant son nom à l'invite de votre système. Si les résultats obtenus sont différents de ceux recherchés, vous devez recommencer à la première étape. Il faut identifier l'origine du problème et corriger le code source. À chaque transformation de ce code, il est nécessaire de recompiler le programme et de relancer l'éditeur de liens (*linker* en anglais) pour créer une version corrigée du fichier exécutable. Répétez ces opérations jusqu'à ce que le programme s'exécute de façon correcte.

Bien que nous ayons différencié la compilation de la liaison, beaucoup de compilateurs exécutent ces deux opérations en une seule étape. Quelle que soit la méthode utilisée, ce sont bien deux actions séparées.

Cycle de développement

Étape 1 Utilisez un éditeur pour créer le code source. Par convention, ce fichier doit avoir l'extension `.c` (par exemple, monprog.c, database.c, etc.).

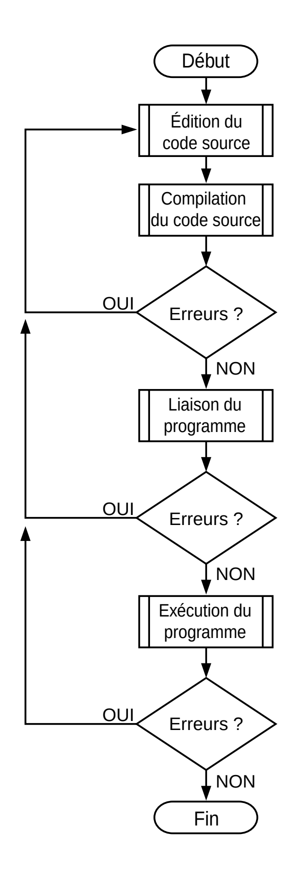
Étape 2 Compilez votre programme. Si le compilateur ne rencontre pas d'erreur dans votre code source, vous obtenez un fichier objet du même nom que votre fichier source avec une extension `.obj` ou `.o` (par exemple, monprog.c est compilé en monprog.o). Si le code source contient des erreurs, le compilateur échoue et vous les affiche pour correction.

Étape 3 Exécutez la liaison. Si aucune erreur n'apparaît, vous obtenez un programme exécutable dans un fichier du même nom que le fichier objet (avec une extension `.exe` sur Windows par exemple, monprog.obj devient monprog.exe).

Étape 4 Exécutez votre programme. Contrôlez les résultats obtenus et recommencez à l'étape 1 si des modifications sont nécessaires dans le fichier source.

Les étapes de développement du programme sont représentées dans la Figure 1.2. Il faut parcourir ce cycle jusqu'à obtenir le résultat recherché. Même le meilleur programmeur ne peut simplement s'asseoir et écrire un programme complet sans aucune erreur dès la première étape. C'est pourquoi il est important de maîtriser parfaitement ces outils : l'éditeur, le compilateur et l'éditeur de liens.

Figure 1.2
Les étapes de développement d'un programme C.



Votre premier programme C

Voici un exemple qui permettra de vous familiariser avec votre compilateur. Même si vous ne comprenez pas la syntaxe, cet exercice est là pour vous faire écrire, compiler, et exécuter un programme C.

Ce programme s'appelle `hello.c`, et il va afficher "Hello, world !" sur votre écran. Vous trouverez le code source de ce programme dans le Listing 1.1. Attention, vous ne devez pas ajouter les numéros de ligne ni les deux points qui suivent. Nous les avons ajoutés dans ce livre pour pouvoir donner la référence des lignes qui seront commentées.

Listing 1.1 : hello.c

```
#include <stdio.h>

int main()
{
   printf("Hello, World !\n");
   return 0;
}
```

Installez votre compilateur en suivant les instructions fournies avec le produit. Quel que soit votre système d'exploitation (Windows, Linux, etc.), assurez-vous d'avoir bien compris le fonctionnement du compilateur et de l'éditeur de votre choix. Vous pouvez maintenant suivre les étapes ci-après pour saisir, compiler, et exécuter hello.c.

Création et compilation de hello.c

Voici comment créer et compiler le programme hello.c :

1. Placez-vous sur le répertoire qui contient vos programmes C et démarrez votre éditeur. Comme nous l'avons mentionné précédemment, vous pouvez utiliser l'éditeur de votre choix. Cependant, beaucoup de compilateurs C (comme anjuta ou Kdevelop sur Linux et visual C/C++ de Microsoft) sont livrés avec un environnement de développement intégré (EDI) qui permet de créer, de compiler et d'effectuer la liaison de façon très conviviale. Consultez vos manuels pour savoir si vous possédez un tel environnement.
2. Utilisez le clavier pour saisir le code source hello.c comme indiqué dans le Listing 1.1 en appuyant sur Entrée à la fin de chaque ligne.



Les numéros de ligne de notre exemple ont été ajoutés pour une meilleure compréhension. Vous ne devez pas les introduire dans votre source.

3. Sauvegardez votre fichier source sous le nom hello.c.
4. Vérifiez que le fichier se trouve bien dans votre répertoire.
5. Exécutez la commande appropriée pour la compilation et la liaison de hello.c.
6. Contrôlez les messages envoyés par le compilateur. Si vous n'avez reçu aucun message d'erreur ou warning, votre fichier source est bon.

   Remarque : Si vous avez fait une erreur de frappe dans votre programme, comme taper `prntf` pour `printf`, le compilateur vous enverra un message comme celui-ci :

   ```
   Error: undefined symbols: prntf in hello.c (hello.OBJ)
   ```

7. Retournez à l'étape 2 si vous avez un message d'erreur. Éditez le fichier hello.c pour comparer son contenu avec Listing 1.1. Faites les corrections nécessaires puis passez à l'étape 3.

8. Votre premier programme C est maintenant prêt à être exécuté. Si vous faites une liste de tous les fichiers de votre répertoire qui s'appellent hello, vous allez voir apparaître :

```
hello.c qui est le fichier source que vous avez créé.
hello.obj ou hello.o qui contient le code objet de hello.c.
hello.exe ou tout simplement hello qui est le programme exécutable,
résultat de la compilation et de la liaison.
```

9. Pour exécuter hello ou hello.exe, entrez simplement `hello`. Le message "Hello, world !" apparaît à l'écran.

Félicitations ! Vous venez de créer, de compiler et d'exécuter votre premier programme C.

Les erreurs de compilation

Une erreur de compilation apparaît lorsque le compilateur rencontre du code source qu'il ne peut pas compiler. Heureusement, les compilateurs d'aujourd'hui vous indiquent la nature et l'emplacement des erreurs pour faciliter la correction du code source.

Cela peut être illustré en introduisant délibérément une erreur dans hello.c. Éditez ce fichier et effacez le point-virgule à la fin de la ligne 5. hello.c ressemble maintenant au fichier du Listing 1.2.

Listing 1.2 : hello.c avec une erreur

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:    printf("Hello, World!")
6:    return 0;
7: }
```

Sauvegardez votre fichier et compilez-le. Votre compilateur va vous envoyer un message qui ressemble à celui-ci :

```
hello.c(6) : Error: ';' expected
```

Vous pouvez remarquer que cette ligne comporte trois parties :

```
hello.c             Le nom du fichier dans lequel se trouve l'erreur
(6) :               Le numéro de la ligne où a été détectée l'erreur
Error: ';' expected Un descriptif de cette erreur
```

Le message vous indique qu'à la ligne 6 de hello, le compilateur n'a pas trouvé de point-virgule. Cette réponse étonnante vient du fait que le point-virgule que vous avez effacé ligne 5 aurait pu se trouver à la ligne suivante (même si ce n'est pas une bonne méthode de

programmation). Ce n'est qu'en contrôlant la ligne 6 que le compilateur a constaté l'absence de point-virgule.

Cela illustre l'ambiguïté des messages d'erreur des compilateurs C. Vous devrez utiliser votre connaissance du langage C pour interpréter ces messages. Les erreurs sont souvent sur la ligne indiquée ou sur celle qui la précède.



Les messages d'erreur peuvent différer d'un compilateur à l'autre. Dans la plupart des cas, les indications qu'il vous fournira vous donneront une bonne idée du problème et de son emplacement.

Avant de continuer notre étude, considérons un autre exemple d'erreur de compilation. Éditez hello.c et transformez-le comme indiqué :

1. Replacez le point-virgule à la fin de la ligne 5.
2. Effacez les guillemets juste avant le mot `Hello`.

Sauvegardez le fichier et compilez de nouveau le programme. Le message d'erreur du compilateur devient :

```
hello.c(5) : Error: undefined identifier "Hello"
hello.c(7) : Lexical error: unterminated string
Lexical error: unterminated string
Lexical error: unterminated string
Fatal error: premature end of source file
```

Le premier message annonce effectivement une erreur en ligne 5 au mot `Hello`. Le message `defined identifier` signifie que le compilateur n'a pas compris `Hello` parce qu'il ne se trouve pas entre guillemets. Les messages suivants, dont nous ne nous préoccuperons pas pour le moment, illustrent le fait qu'une seule erreur dans un programme C peut quelquefois provoquer de multiples messages.

Voici ce que vous devrez en retenir : si le compilateur vous envoie plusieurs messages d'erreur, et que vous n'en trouviez qu'une, corrigez-la et recompilez votre programme. Cette seule correction pourrait annuler tous les messages.

Les messages d'erreur de l'éditeur de liens

Des erreurs en provenance de l'éditeur de liens sont relativement rares et sont généralement dues à une faute de frappe dans le nom d'une fonction appartenant à la bibliothèque C. Dans ce cas, le message suivant apparaît : `Error: undefined symbols: error message`, suivi du nom mal orthographié (précédé d'un tiret).

Résumé

La lecture de ce premier chapitre vous a certainement convaincu que le choix du C comme langage de programmation est judicieux. Il offre une bonne combinaison entre puissance, portabilité et notoriété. À ces qualités s'ajoute la possibilité d'évoluer vers le langage orienté objet C++ ou Java.

Ce chapitre a décrit les différentes étapes du développement d'un programme C. Vous devez maîtriser le cycle édition-compilation-liaison-tests ainsi que les outils nécessaires à chaque étape.

Les erreurs sont indissociables du développement d'un programme. Votre compilateur les détecte et vous envoie un message d'erreur qui en donne la nature et l'emplacement. Ces informations permettent d'éditer le code source pour le corriger. Rappelez-vous cependant que ces messages ne sont pas toujours très précis, et qu'il faut utiliser votre connaissance du C pour les interpréter.

Q & R

Q Si je veux donner mon programme à quelqu'un, de quels fichiers a-t-il besoin ?

R Le fait que le langage C soit un langage compilé est un avantage. Cela signifie que lorsque votre code source est compilé, vous obtenez un programme exécutable qui se suffit à lui-même. Pour donner Hello à tous vos amis, il suffit de leur donner l'exécutable (hello ou hello.exe). Ils n'ont pas besoin du fichier source hello.c, ni du fichier objet hello.o ou hello.obj. Ils n'ont pas besoin non plus de posséder un compilateur C. Néanmoins, diffuser les sources (hello.c) accompagnées d'une licence libre permettra à vos amis développeurs d'améliorer Hello.

Q Faut-il conserver les fichiers sources (.c) et objets (.obj ou .o) après la création du fichier exécutable ?

R Si vous supprimez votre fichier source, vous n'aurez aucune possibilité plus tard d'apporter une modification à votre programme. Vous devriez donc le garder. Pour ce qui concerne le fichier objet, vous pouvez en obtenir une copie à tout moment en recompilant le fichier source. Vous n'avez donc pas besoin de le conserver.

La plupart des environnements de développement intégrés créent des fichiers qui s'ajoutent à ceux déjà cités ci-avant. Vous pourrez les recréer aussi longtemps que vous serez en possession du fichier source (.c).

Q Faut-il utiliser l'éditeur qui est livré avec le compilateur ?

R Ce n'est pas une obligation. Vous pouvez utiliser l'éditeur de votre choix du moment que vous sauvegardez le code source en format texte. Si votre compilateur possède un éditeur, vous devriez l'essayer et choisir celui qui vous convient le mieux. J'utilise moi-même un éditeur que j'ai acheté séparément alors que tous les compilateurs que j'utilise en ont un. Ces éditeurs qui sont livrés avec les compilateurs sont de plus en plus performants. Certains formatent automatiquement votre code source. D'autres distinguent les différentes parties de votre fichier source à l'aide de couleurs différentes pour vous aider à trouver les erreurs.

Q Peut-on ignorer les messages d'avertissement ?

R Certains de ces messages n'affectent en rien l'exécution de votre programme, d'autres pas. Si votre compilateur vous envoie un message de warning, cela signale que quelque chose ne convient pas. Beaucoup de compilateurs vous offrent la possibilité de supprimer ce genre de message en dessous d'un certain niveau. Le compilateur ne donnera que les messages les plus sérieux. Vous devriez cependant consulter tous vos messages, un programme est meilleur s'il ne comporte aucune erreur ou warning (le compilateur ne produit pas de programme exécutable s'il reste une seule erreur dans le source).

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre. Essayez de comprendre les réponses fournies dans l'Annexe G avant de passer au chapitre suivant.

Quiz

1. Donnez trois raisons pour lesquelles le C est un bon choix de langage de programmation.
2. Quel est le rôle du compilateur ?
3. Quelles sont les étapes du cycle de développement d'un programme ?
4. Quelle commande permet de compiler le programme program.c ?
5. Votre compilateur exécute-t-il la compilation et la liaison avec la même commande ?
6. Quelle extension devriez-vous utiliser pour votre fichier source C ?
7. Est-ce que filename.txt est un nom correct pour votre fichier source C ?

8. Que faut-il faire si le programme que vous avez compilé ne donne pas les résultats escomptés ?
9. Qu'est ce que le langage machine ?
10. Que fait l'éditeur de liens ?

Exercices

1. Éditez le fichier objet créé avec Listing 1.1. Ressemble-t-il au fichier source ? (Ne sauvegardez pas ce fichier lorsque vous quitterez l'éditeur.)
2. Entrez le programme suivant et compilez-le. Que fait-il ? (Ne saisissez pas les numéros de ligne ni les deux points.)

```
 1: #include <stdio.h>
 2:
 3: int rayon, aire;
 4:
 5: int main()
 6: {
 7:    printf("Entrez le rayon (ex 10) : ");
 8:    scanf("%d", &rayon);
 9:    aire =  (3.14159 * rayon * rayon);

10:   printf("\n\nAire = %d\n", aire);
11:   return 0;
12: }
```

3. Saisissez et compilez le programme suivant. Que fait-il ?

```
 1: #include <stdio.h>
 2:
 3: int x,y;
 4:
 5: int main()
 6: {
 7:    for (x = 0; x < 10; x++, printf("\n"))
 8:    for (y = 0; y < 10; y++)
 9:    printf ("X");
10:
11:   return 0;
12: }
```

4. **CHERCHEZ L'ERREUR** : Saisissez ce programme et compilez-le. Quelles sont les lignes qui génèrent des erreurs ?

```
1: #include <stdio.h>
2:
3: int main();
4: {
```

```
5:    printf ("Regardez bien !");
6:    printf ("Vous allez trouver !");
7:    return 0;
8: }
```

5. **CHERCHEZ L'ERREUR** : Saisissez ce programme et compilez-le. Quelles sont les lignes qui génèrent des erreurs ?

```
1: #include <stdio.h>
2:
3: int main();
4: {
5:    printf ("Ce programme a vraiment ");
6:    do_it("un problem !");
7:    return 0;
8: }
```

6. Transformez la ligne 9 de l'exercice 3 comme indiqué. Recompilez et exécutez le nouveau programme. Que fait-il maintenant ?

```
9:  printf("%c", 65);
```

Exemple pratique 1

Lecture au clavier et affichage à l'écran

Vous trouverez, dans ce livre, plusieurs sections de ce type présentant un programme un peu plus long que les exemples fournis dans les chapitres. Il pourra contenir des éléments qui n'auront pas encore été abordés, mais vous aurez ainsi la possibilité de saisir un programme complet puis de l'exécuter.

Les programmes présentés constitueront des applications pratiques ou amusantes. Le programme `perroquet` de cette section, par exemple, lit une ligne au clavier et l'affiche. Ce programme contient d'ailleurs une fonction qui sera utilisée tout au long de cet ouvrage : lire_clavier(). Vous devrez la recopier telle quelle dans chaque programme qui y fait appel.

Prenez le temps de tester ces programmes. Modifiez-les, recompilez, puis exécutez-les de nouveau. Observez les résultats ainsi obtenus. Nous n'expliquerons pas les détails de fonctionnement au niveau du code, seulement les opérations effectuées. Vous en comprendrez toutes les subtilités lorsque vous aurez parcouru tous les chapitres. Vous avez ainsi la possibilité d'aborder rapidement des programmes intéressants.

Le premier exemple pratique

Saisissez et compilez le programme suivant, en prenant soin de ne pas introduire de fautes de frappe (vous les retrouverez sous la forme d'erreurs au moment de la compilation).

Pour exécuter ce programme, tapez `perroquet`. Ne soyez pas impressionné par sa longueur, vous n'êtes pas censé comprendre chaque ligne de code pour l'instant.

Listing Exemple pratique 1 : perroquet.c

```c
/* perroquet.c : ce programme répète ce qu'il vient de lire au clavier */
#include <stdlib.h>
#include <stdio.h>

int lire_clavier(char *str, int taille)
{
    int i;
    fgets(str, taille, stdin);
    str[taille-1] = '\0';
    for(i=0; str[i]; i++) /* supprime le retour chariot */
    {
        if(str[i] == '\n')
        {
            str[i] = '\0';
            break;
        }
    }
    return(i); /* Renvoie 0 si la chaîne est vide */
}

int main()
{
    char buffer[80];

    printf("Entrez une ligne et validez avec Entrée\n");
    lire_clavier(buffer, sizeof(buffer));
    printf("Vous avez écrit : '%s'\n", buffer)

    exit(EXIT_SUCCESS);
}
```

Le Chapitre 5 sur les fonctions, ainsi que le Chapitre 14, qui traite des entrées/sorties, vous aideront à comprendre le fonctionnement de ce programme.

2

Structure d'un programme C

Un programme C est constitué de plusieurs modules de programmation ou blocs. Une grande partie de ce livre traite de ces divers éléments de programme et de leur utilisation. Avant de détailler chacun d'eux, nous allons étudier un programme C complet.

Aujourd'hui, vous allez apprendre à :

- Identifier les blocs de programme à partir d'un exemple simple
- Reconnaître les caractéristiques de chacun de ces blocs
- Compiler et exécuter un programme de test

Exemple de programme

Le Listing 2.2 représente le code source du programme de test. Ce programme est très simple : il donne le produit de deux nombres saisis au clavier. N'essayez pas d'en comprendre les détails, ce chapitre est destiné à vous familiariser avec les composants d'un programme C.

Avant d'étudier votre programme de test, vous devez savoir ce que représente une *fonction*. C'est une partie indépendante du code du programme qui effectue une certaine tâche, et qui est référencée par un nom. En introduisant ce nom dans le programme, celui-ci peut exécuter le code qui lui est associé. Le programme peut transmettre des informations, appelées arguments, à cette fonction qui pourra à son tour lui renvoyer une valeur. Les deux types de fonctions C sont les *fonctions de bibliothèque*, qui sont fournies avec le compilateur C, et les *fonctions utilisateur*, que le programmeur peut lui-même créer.

Nous vous rappelons que les numéros de ligne inclus dans cet exemple, comme dans tous les exemples de ce livre, ne font pas partie du programme. Ne les tapez pas.

Listing 2.1 : multiplier.c

```
/* Calcul du produit de deux nombres. */
#include <stdio.h>

int produit(int x, int y);


int main()
{
     int a,b,c;

/* Lecture du premier nombre */
    printf("Entrez un nombre entre 1 et 100 : ");
    scanf("%d", &a);

/* Lecture du deuxième nombre */
   printf("Entrez un autre nombre entre 1 et 100 : ");
   scanf("%d", &b);

/* Calcul du produit et affichage du résultat */
   c = produit(a, b);
   printf ("\n%d fois %d = %d", a, b, c);

   return 0;
}

```

```
25: /* La fonction renvoie le produit de ses deux arguments */
26: int produit(int x, int y)
27: {
28:    return (x * y);
29: }
```

```
Entrez un nombre entre 1 et 100 : 35
Entrez un autre nombre entre 1 et 100 : 23

35 fois 23 = 805
```

Structure du programme

Nous allons examiner le programme précédent ligne par ligne pour en isoler les différents composants.

La fonction *main()*

La fonction `main()` est le seul bloc obligatoire d'un programme C. Sa forme la plus simple consiste à saisir son nom, `main`, suivi de parenthèses `()` vides et d'une paire d'accolades`{}`. Celles-ci renferment la partie principale du programme. L'exécution du programme débute à la première instruction de `main()` et se termine avec la dernière instruction de cette fonction.

Appel d'un fichier *#include*

L'instruction d'appel `#include`, indique au compilateur C qu'il doit inclure le contenu d'un fichier dans le programme pendant la compilation. Ce fichier *inclus* (aussi appelé *fichier en-tête*) contient des informations destinées à votre programme ou au compilateur. Plusieurs de ces fichiers   été livrés avec votre compilateur, vous ne devez pas en modifier les informations. Ils ont tous une extension .h (par exemple, stdio.h).

Dans notre exemple, l'instruction d'appel `#include` signifie "ajouter le contenu du fichier stdio.h". Le Chapitre 21 vous donnera de plus amples informations sur ces fichiers.

La définition de variable

Une *variable* est un nom donné à une zone mémoire. En effet, votre programme a besoin de mémoire pour stocker ses données en cours d'exécution. En C, une variable doit être définie avant d'être utilisée. La *définition de variable* indique son nom au compilateur et le type de données que l'on pourra y stocker. La définition de la ligne 8 de notre exemple, `int a, b, c;`, définit trois variables appelées `a`, `b`, et `c` qui contiendront chacune une *valeur entière*. Les variables et constantes numériques sont traitées au Chapitre 3.

La déclaration de fonction

La *déclaration de fonction* indique au compilateur C le nom et les arguments d'une fonction qui sont utilisés dans le programme. Cette déclaration doit apparaître avant l'utilisation de la fonction et ne doit pas être confondue avec la *définition de fonction* qui contient les instructions propres à cette fonction. Cette déclaration est facultative si la fonction peut être définie avant tout appel à elle.

Les instructions

Les *instructions* constituent le travail réalisé par le programme. Elles affichent les informations sur l'écran, lisent les données saisies au clavier, effectuent les opérations mathématiques, appellent les fonctions, lisent les fichiers et accomplissent tous types d'opérations nécessaires à un programme. Chaque instruction occupe généralement une ligne et se termine par un point-virgule. Ce livre est consacré en grande partie à l'enseignement de ces différentes instructions.

printf ()

`printf()` (lignes 11, 15, et 20) est une fonction de bibliothèque qui envoie des informations à l'écran. Elle peut afficher un message texte simple (comme en ligne 11 ou 15) ou un message accompagné de variables issues du programme (comme en ligne 20).

scanf ()

`scanf()` (lignes 12 et 16) est une autre fonction de bibliothèque. Elle lit les données entrées au clavier et les attribue à des variables du programme.

L'instruction de la ligne 19 *appelle* la fonction `produit()` et lui transmet les *arguments* `a` et `b`. Le programme exécute alors les instructions appartenant à la fonction `produit()` qui lui renvoie une valeur. Cette valeur est sauvegardée dans la variable c.

return

L'instruction `return` de la ligne 28 fait partie de la fonction `produit()`. Elle calcule le produit des variables `x` et `y`, puis renvoie le résultat au programme appelant.

La définition de fonction

Une *fonction* est une portion de code indépendante qui a été écrite pour effectuer une certaine tâche. On *appelle* cette fonction dans un programme en introduisant son nom dans une instruction.

La fonction `produit()`, jusqu'à la ligne 29, est une *fonction utilisateur* . Comme son nom l'indique, une fonction utilisateur est écrite par le programmeur pendant le développement de son programme. Celle-ci est simple, elle multiplie deux valeurs et renvoie la réponse au programme qui l'a appelée. Vous apprendrez au Chapitre 5 qu'une bonne programmation C est basée sur une utilisation correcte de ces fonctions.

En réalité, vous n'avez pas besoin de créer une fonction pour une tâche aussi simple que la multiplication de deux nombres. Nous l'avons fait pour vous donner un exemple.

Le langage C possède de multiples *fonctions de bibliothèques* qui sont fournies avec le compilateur. Ces fonctions réalisent la plupart des tâches de base (comme les entrées/sorties de l'écran, du clavier, et du disque) dont votre programme a besoin. Dans notre exemple, `printf()` et `scanf()` sont des fonctions de bibliothèque.

Les commentaires du programme

La partie de code du programme qui commence par /* et qui se termine par */ est un commentaire. Le compilateur l'ignore. Vous pouvez le placer n'importe où, il n'a aucune influence sur le déroulement du programme. Un commentaire peut s'étendre sur une ou plusieurs lignes, ou sur une partie de ligne seulement. En voici trois exemples :

```
/* un commentaire d'une ligne */

int a, b, c; /* sur une partie de ligne */

/* un commentaire
qui s'étend
sur plusieurs lignes */
```

Vous ne devez pas imbriquer des commentaires, cela provoque une erreur avec beaucoup de compilateurs :

```
/*
/* mauvais exemple à ne pas suivre */
*/
```

Même si certains compilateurs les acceptent, évitez-les si vous voulez conserver une bonne portabilité de votre code C. De tels commentaires peuvent aussi conduire à des problèmes difficiles à résoudre.

Beaucoup d'apprentis programmeurs considèrent les commentaires comme une perte de temps inutile. C'est une erreur ! Votre code peut vous sembler tout à fait clair pendant que vous le développez, surtout s'il s'agit d'un programme simple. Mais s'il évolue dans le temps pour devenir plus complexe, vous apprécierez ces commentaires quand vous aurez à le modifier. Prenez l'habitude de bien documenter ce qui le nécessite, en faisant également attention à ne pas trop en mettre.



Certains programmeurs utilisent un type de commentaire plus récent qui est disponible avec le langage C++ ou Java : le double slash (//). En voici deux exemples :

```
// cette ligne est un commentaire
int x;// les commentaires débutent après les deux slash
```

Les deux slashs signifient que la fin de la ligne est un commentaire. Même si beaucoup de compilateurs les acceptent, vous devriez les éviter pour conserver une bonne portabilité de votre code.



À faire

Commenter votre code source, surtout s'il contient des algorithmes qui pourraient être difficiles à comprendre. Vous gagnerez un temps précieux quand vous aurez à les modifier.

À ne pas faire

Formuler des commentaires inutiles. Par exemple,

```
/* Le programme suivant affiche "Hello, world !" sur votre écran */
printf("Hello, World ! ");
```

Ce commentaire est inutile si vous connaissez le fonctionnement de `printf()`.

À faire

Apprendre à doser les commentaires dans vos programmes. S'ils sont peu nombreux ou en style télégraphique, ils ne seront pas d'un grand secours. S'ils sont trop longs, vous passerez plus de temps à commenter qu'à programmer.

Les accolades

Les accolades ({}) permettent d'encapsuler les lignes de programmes qui constituent chaque fonction C. On appelle *bloc* l'ensemble des instructions qui se trouvent entre ces accolades.

Comment exécuter le programme

Prenez le temps de saisir, compiler, et exécuter multiplier.c. C'est une bonne occasion d'utiliser votre éditeur et votre compilateur. Rappelez-vous les étapes du Chapitre 1 :

1. Placez-vous sur votre répertoire de programmation.
2. Démarrez votre éditeur.
3. Saisissez le code source comme indiqué dans le Listing 2.1, sans les numéros de ligne.

4. Sauvegardez votre programme.
5. Lancez la compilation et la liaison du programme avec les commandes correspondantes de votre compilateur. Si aucun message d'erreur n'apparaît, vous pouvez exécuter le programme en tapant `multiplier` à l'invite du système.
6. Si vous obtenez un message d'erreur, retournez à l'étape 2 et corrigez votre fichier source.

Remarque

Un ordinateur est précis et rapide, mais il ne fait qu'exécuter des ordres. Il est parfaitement incapable de corriger la moindre erreur .

Cela est valable pour votre code source C. Le compilateur échouera à la moindre faute de frappe. Heureusement, même s'il ne peut pas corriger vos erreurs, il sait les reconnaître pour vous les indiquer. (Les messages du compilateur et leur interprétation sont traités dans le chapitre précédent.)

Étude de la structure d'un programme

Vous connaissez maintenant la structure d'un programme. Étudiez le Listing 2.2 et essayez d'en reconnaître les différentes parties.

Listing 2.2 : list_it.c

```
/*list_it.c Ce programme affiche du code source avec les numéros
  de lignes. */
#include <stdio.h>
#include <stdlib.h>

void display_usage(void);

int line;

int main(int argc, char *argv[])
{
   char buffer[256];
   FILE *fp;

   if(argc < 2)
   {
       display_usage();
       exit(EXIT_FAILURE);
   }

   if ((fp = fopen(argv[1], "r")) == NULL)
   {
       fprintf(stderr, "erreur fichier, %s!", argv[1]);
```

```
        exit(EXIT_FAILURE);
    }

    line = 1;

    while(lire_clavier(buffer, sizeof(buffer)))
        fprintf(stdout, "%4d:\t%s", line++, buffer);

    fclose(fp);
    exit(EXIT_SUCCESS);
}

void display_usage(void)
{
    fprintf(stderr, "La syntaxe est la suivante :\n\n");
    fprintf(stderr, "list_it filename.ext\n");
}
```

```
 C:\>list_it list_it.c
 1: /*list_it.c Ce programme affiche du code source
      avec les numéros de lignes. */
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4:
 5: void display_usage(void);

 6: int line;
 7:
 8: int main(int argc, char *argv[])
 9: {
10:    char buffer[256];
11:    FILE *fp;
12:
13:    if(argc < 2)
14:    {
15:        display_usage();
16:        exit(EXIT_FAILURE);
17:    }
18:
19:    if ((fp = fopen(argv[1], "r")) == NULL)
20:    {
21:        fprintf(stderr, "erreur fichier, %s!", argv[1]);
22:        exit(EXIT_FAILURE);
23:    }
24:
25:    line = 1;
26:
27:    while(lire_clavier(buffer, sizeof(buffer)))
28:        fprintf(stdout, "%4d:\t%s", line++, buffer);
29:
30:    fclose(fp);
31:    exit(EXIT_SUCCESS);
32: }
33:
34: void display_usage(void)
35: {
36:    fprintf(stderr, "\nLa syntaxe est la suivante : ");
37:    fprintf(stderr, "\n\nLIST_IT filename.ext\n");
38: }
```

Analyse

list_it.c ressemble à print_it.c du Chapitre 1. Il permet d'afficher le source du programme numéroté à l'écran, au lieu de l'envoyer vers l'imprimante.

Nous pouvons identifier les différentes parties de ce programme. La fonction `main()` est développée de la ligne 8 à 32. Les lignes 2 et 3 contiennent les appels du fichier en-tête `#include` et les définitions de variables sont en lignes 6, 10 et 11. Nous trouvons la déclaration de fonction, `void display usage(void)`, en ligne 5. Ce programme possède de nombreuses instructions (lignes 13, 15, 16, 19, 21, 22, 25, 27, 28, 30, 31, 36, et 37). Les lignes 34 à 38 représentent la définition de fonction `display usage`. La ligne 1 est une ligne de commentaires et des accolades séparent les différents blocs du programme.

list_it.c appelle plusieurs fonctions. Les fonctions de bibliothèque utilisées sont `exit()` en lignes 16, 22 et 31, `fopen()` en ligne 19, `fprintf()` en lignes 21, 28, 36, et 37, `lire clavier()` en ligne 27 (notre fonction définie dans l'exemple pratique 1), enfin `fclose()` en ligne 30. `display usage()` est une fonction utilisateur. Toutes ces fonctions sont traitées plus loin dans ce livre.

Résumé

Ce chapitre court a abordé un sujet important : les principaux composants d'un programme C. Vous avez appris que la fonction `main()` est obligatoire et que les instructions du programme permettent de transmettre vos ordres à l'ordinateur. Ce chapitre a aussi introduit les variables, leurs définitions, et vous a expliqué comment et pourquoi introduire des commentaires dans le code source.

Un programme C peut utiliser deux types de fonctions : les fonctions de bibliothèque qui sont fournies avec le compilateur, et les fonctions utilisateur qui sont créées par le programmeur.

Q & R

Q Les commentaires ont-ils une influence sur le déroulement du programme ?

R Les commentaires sont destinés aux programmeurs. Lorsque le compilateur converti le code source en code objet, il supprime tous les blancs et commentaires. Ils n'ont donc aucune influence sur l'exécution du programme. Les blancs et commentaires permettent simplement de clarifier le code source pour faciliter la lecture et la maintenance du programme.

Q Quelle est la différence entre une instruction et un bloc ?

R Un bloc est constitué d'un groupe d'instructions entre accolades (`{}`).

Q Comment puis-je connaître les fonctions de bibliothèque disponibles ?

R Beaucoup de compilateurs sont livrés avec un manuel contenant toutes les fonctions de bibliothèque. Elles sont généralement classées par ordre alphabétique. L'Annexe E énumère une grande partie des fonctions disponibles. Consultez-la avant de programmer, cela vous épargnera de créer des fonctions qui existent déjà dans la bibliothèque.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre. Essayez de comprendre les réponses fournies dans l'Annexe G avant de passer au chapitre suivant.

Quiz

1. Comment appelle-t-on un groupe d'une ou plusieurs instructions entre accolades ?
2. Quel est l'élément obligatoire d'un programme C ?
3. Comment peut-t-on introduire des commentaires dans un programme ? Pour quelle raison doit-on documenter les programmes ?
4. Qu'est-ce qu'une fonction ?
5. Quels sont les deux types de fonctions disponibles en langage C et quelles sont leurs différences ?
6. À quoi sert l'appel `#include` ?
7. Peut-on imbriquer des commentaires ?
8. Peut-on faire des commentaires sur plus d'une ligne ?
9. Quel est l'autre nom d'un fichier inclus ?
10. Qu'est-ce qu'un fichier inclus ?

Exercice

1. Écrivez le programme le plus court possible.
2. Étudiez le programme suivant :

```
/* ex2-2.c */
#include <stdio.h>

void display_line(void);

int main()
{
   display_line();
   printf("\n Le langage C en 21 jours !\n");
```

```
10:   display_line();
11:
12:   exit(EXIT_SUCCESS);
13: }
14:
15: /* Affichage d'une ligne d'asterisques */
16: void display_line(void)
17: {
18:   int counter;
19:
20:   for(counter = 0; counter < 21; counter++)
21:    printf("*");
22: }
```

a) Quelles sont les lignes qui contiennent des instructions ?
b) Dans quelles lignes se situent les définitions de variables ?
c) Quels sont les numéros de ligne des déclarations de fonction ?
d) Quelles lignes contiennent les définitions de fonction ?
e) Quelles sont les lignes qui ont des commentaires ?

3. Écrivez une ligne de commentaires.
4. Que fait le programme suivant ? (saisissez-le, compilez et exécutez-le)

```
/* ex2-4.c */
#include <stdio.h>
#include <string.h>
int main()
{
   int ctr;

   for(ctr = 65; ctr < 91; ctr++)
   printf("%c", ctr);

 exit(EXIT_SUCCESS);
}
/* fin du programme */
```

5. Que fait le programme suivant ? (saisissez-le, compilez, et exécutez-le)

```
/* ex2-5.c */
#include <stdio.h>

int main()
{
   char buffer[256];

   printf("Entrez votre nom et appuyez sur Entrée:\n");
   lire_clavier(buffer, sizeof(buffer));

  printf("\nVotre nom contient %d caractères.", strlen(buffer));

  exit(EXIT_SUCCESS);
}
```

3

Constantes et variables numériques

Les programmes d'ordinateur travaillent avec différents types de données et ont besoin de mémoire pour les stocker. Le langage C peut stocker des données sous formes de variable ou de constante avec de multiples options. Une *variable* dispose d'une zone de stockage en mémoire et sa valeur peut changer en cours de programme. Une *constante*, au contraire, contient une valeur fixe.

Aujourd'hui, vous allez apprendre :

- Comment créer un nom de variable
- Comment utiliser les différents types de variable numérique
- Les différences entre caractères et valeurs numériques
- Comment déclarer et initialiser les variables numériques
- Quels sont les deux types de constantes numériques du langage C

Avant d'aborder les variables, vous devez connaître les principes de fonctionnement de la mémoire de votre ordinateur.

La mémoire

Si vous savez déjà comment fonctionne la mémoire de votre ordinateur, vous pouvez passer au paragraphe suivant. Les informations qui suivent vont permettre de mieux comprendre certains aspects de la programmation C.

Un ordinateur utilise de la mémoire vive (RAM, *Random Access Memory*) pour stocker des informations pendant son fonctionnement. Cette mémoire se situe dans une puce à l'intérieur de votre ordinateur. La mémoire vive est *volatile*, ce qui signifie qu'elle est allouée ou libérée pour de nouvelles informations aussi souvent que nécessaire. Cela signifie aussi qu'elle ne fonctionne que lorsque l'ordinateur est sous tension. Lorsque vous le débranchez, vous perdez toutes les informations qui s'y trouvaient.

La quantité de mémoire vive installée sur chaque ordinateur est variable. On l'exprime en multiples d'octets (mégaoctets ou gigaoctets). Si autrefois la mémoire était comptée, nos ordinateurs disposent aujourd'hui de mégaoctets, voire de gigaoctets, et les programmeurs ont la fâcheuse tendance à l'utiliser sans compter.

L'octet est l'unité de base de la mémoire ordinateur. Le Chapitre 20 traite de cette notion d'octet. Le Tableau 3.1 vous donne quelques exemples du nombre d'octets nécessaires pour stocker différentes sortes de données.

Tableau 3.1 : Exemples de tailles mémoire

Donnée	***Nombre d'octets nécessaires***
Le caractère *x*	1
Le nombre 500	2
Le nombre 241,105	4
La phrase "j'apprends le C"	25
Une page de manuel	Environ 3 000

La mémoire RAM est sollicitée de façon séquentielle et chaque octet est identifié par une *adresse* unique. Cette adresse commence à zéro, pour le premier octet de mémoire, et s'incrémente à chaque octet en séquence jusqu'à la limite du système. Ces adresses sont gérées automatiquement par votre compilateur.

La mémoire vive a plusieurs fonctions, mais celle qui vous intéresse en tant que programmeur est le stockage des données. Quelle que soit la tâche réalisée par votre programme, il

travaille avec des *données* qui sont stockées dans la mémoire vive de votre ordinateur pendant toute la durée de l'exécution.

Maintenant que vous connaissez ses principes de fonctionnement, nous pouvons étudier comment le langage C utilise cette mémoire pour stocker ses informations.

Les variables

Une *variable* est le nom d'une zone mémoire de votre ordinateur. En utilisant ce nom dans votre programme, vous adressez la donnée qui y est stockée.

Les noms de variable

Avant d'utiliser une variable dans votre programme, vous devez créer un nom de variable qui doit respecter plusieurs règles :

- Ce nom peut contenir des lettres, des chiffres et le caractère ( ).
- Le premier caractère doit être une lettre. Le caractère ( ) est aussi autorisé, mais il n'est pas recommandé.
- Les lettres majuscules sont différentes des minuscules. Par exemple, `compte` et `Compte` ne représentent pas la même variable.
- Il ne faut pas utiliser les mots clés, ils font partie du langage C. L'Annexe B fournit une liste complète des 33 mots clés du C.

Voici quelques exemples de noms de variables C :

Nom de la variable	***Validité***
`Pourcent`	Correct
`y2x5 fg7h`	Correct
`profit annuel`	Correct
` taxe 1990`	Correct, mais déconseillé
`compte#courant`	Incorrect : contient le caractère interdit #
`double`	Incorrect : double est un mot clé
`9puits`	Incorrect : le 1er caractère est un chiffre

Certains compilateurs ne considèrent que les 31 premiers caractères d'un nom de variable, même si celui-ci peut être plus long. Cela permet de donner des noms qui reflètent le

type de donnée qui y est sauvegardé. Par exemple, un programme qui calcule des échéances de prêt pourrait stocker la valeur du taux d’intérêt dans une variable appelée `taux interets`. Son utilisation en devient plus aisée et le programme sera plus facile à lire et à comprendre.

Il existe de nombreuses conventions pour ces noms de variables. Nous venons d’en voir un exemple en utilisant le caractère ( ) pour séparer des mots à l’intérieur du nom de la variable. Comme nous l’avons vu, cette séparation permet de l’interpréter plus facilement. Une seconde solution consiste à remplacer l’espace par une lettre majuscule. Notre exemple précédent `taux interets`, deviendrait `TauxInterets`. Cette notation est de plus en plus répandue parce qu’il est plus facile de taper une lettre majuscule que le caractère ( ). Nous l’avons utilisé dans ce livre parce que la lecture en est plus facile.



À faire

Utiliser des noms de variables mnémotechniques.

Se fixer une convention pour les noms de variables.

À ne pas faire

Ne pas faire précéder les noms de variables du caractère ( ), ou les écrire en lettres majuscules alors que ce n’est pas nécessaire.

Les types de variables numériques

Il existe en C plusieurs types de variable numérique. Leurs différences s’expliquent par le fait que des valeurs numériques selon leur taille ont des besoins de mémoire différents et que les opérations mathématiques ne s’effectuent pas de la même façon selon le type de variables. Les petits entiers (par exemple 1, 199 et 8) demandent peu d’espace mémoire pour être stockés et les opérations mathématiques (additions, multiplications, etc.) sont réalisées très rapidement par votre ordinateur. Les grands nombres et les valeurs en virgule flottante (123 000 000 ou 0,000000871256, par exemple), au contraire, nécessitent plus d’espace mémoire et les opérations mathématiques sont plus longues. En utilisant le type de variables approprié, vous optimisez l’exécution de votre programme.

Voici les deux principales catégories de variables numériques C :

- Les *variables entières* qui sont un nombre entier positif ou négatif, ou le zéro.
- Les *variables à virgule flottante* qui contiennent des valeurs pouvant avoir des chiffres après la virgule (ce sont les nombres réels).

Chacune de ces catégories se divise en plusieurs types de variables. Le Tableau 3.2 récapitule ces différents types et vous donne l'espace mémoire nécessaire pour stocker chacune de ces variables si vous utilisez un micro-ordinateur à architecture 16 bits.

Tableau 3.2 : Les types de données numériques en C (représentation ILP32)

Type de variable	*Mot clé*	*Octets nécessaires*	*Intervalle des valeurs*
Caractère	`char`	1	–128 à 127
Entier court	`short`	2	– 32 768 à 32 767
Entier	`int`	4	–2 147 483 648 à 2 147 438 647
Entier long	`long`	4	–2 147 483 648 à 2 147 438 647
Caractère non signé	`unsigned char`	1	0 à 255
Entier court non signé	`unsigned short`	2	0 à 65 535
Entier non signé	`unsigned int`	4	0 à 4 294 967 295
Entier long non signé	`unsigned long`	4	0 à 4 294 967 295
Simple précision virgule flottante	`float`	4	1,2 E–38 à 3,4 E38 *
Double précision virgule flottante	`double`	8	2,2 E–308 à 1,8 E308 **

* Valeur approximative : précision = 7 chiffres.
** Valeur approximative : précision = 19 chiffres

Valeur approximative signifie la plus haute et la plus petite valeur qu'une variable donnée puisse recevoir. `Précision` indique la précision avec laquelle la variable est stockée (par exemple, pour évaluer 1/3, la réponse est 0,333 avec des 3 à l'infini ; une variable avec une précision de 7 s'écrira avec sept chiffres 3 après la virgule).

Vous pouvez remarquer que dans le Tableau 3.2 les types de variables `int` et `long` sont identiques. Cela est vrai sur des systèmes compatibles PC en 32 bits, en représentation ILP32 mais ces deux variables peuvent être différentes sur d'autres types de matériels. Sur un ancien système 16 bits, long et `int` n'ont pas la même taille. La taille de `short` est de 2 octets, alors que celle de `int` est de 4. La portabilité du langage C exige donc deux mots clés différents pour ces deux types.

Les variables entières sont des nombres réels par défaut, elles n'ont pas de mot clé particulier. Vous pouvez toutefois inclure le mot clé `signed` si vous le désirez. Les mots clés du

Tableau 3.2 sont utilisés dans les déclarations de variable, qui sont traitées dans le prochain paragraphe.

Le Listing 3.1 va permettre de connaître la taille des variables sur votre ordinateur. Ne soyez pas surpris si vos résultats sont différents de ceux présentés ci-après.

Listing 3.1 : Ce programme affiche la taille des types de variables

```
/* sizeof.c  Ce programme vous donne la taille des types */
/*           variables C  en octets */

#include <stdio.h>

int main()
{

   printf("\n char a une taille de %d octets", sizeof(char));
   printf("\n int a une taille de %d octets", sizeof(int));
   printf("\n short a une taille de %d octets", sizeof(short));
   printf("\n long a une taille de %d octets", sizeof(long));
   printf("\n unsigned char a une taille de %d octets",
   sizeof(unsigned char));
   printf("\n unsigned int a une taille de %d octets",
   sizeof(unsigned int));
   printf("\n unsigned short a une taille de %d octets",
   sizeof(unsigned short));
   printf("\n unsigned long a une taille de %d octets",
   sizeof(unsigned long));
   printf("\n float a une taille de %d octets",sizeof(float));
   printf("\n double a une taille de %d octets\n",sizeof(double));

   exit(EXIT_SUCCESS);
}
```

```
char             a une taille de 1 octets
int              a une taille de 4 octets
short            a une taille de 2 octets
long             a une taille de 4 octets
unsigned char    a une taille de 1 octets
unsigned int     a une taille de 4 octets
unsigned short   a une taille de 2 octets
unsigned long    a une taille de 4 octets
float            a une taille de 4 octets
double           a une taille de 8 octets
```

Analyse

Vous connaissez maintenant la taille de chaque type de variable sur votre ordinateur. Si vous utilisez un PC en mode 32 bits, vos chiffres devraient correspondre à ceux du Tableau 3.2.

Certaines parties de ce programme doivent vous sembler familières. Les lignes 1 et 2 sont des commentaires avec le nom du programme et une brève description. La ligne 4 appelle le fichier en-tête standard pour l'affichage des informations à l'écran. Ce programme ne contient que la fonction principale `main()` en lignes 7 à 25. Les lignes 9 à 22 affichent la taille de chaque type de variable à l'aide de l'opérateur `sizeof` (voir Chapitre 19). La ligne 24 du programme renvoie la valeur EXIT_SUCCESS au système d'exploitation avant la fin de l'exécution du programme.

Voici les caractéristiques imposées par la norme ANSI :

- La taille d'un caractère est d'un octet.
- La taille d'une variable `short` est inférieure ou égale à celle d'une variable `int`.
- La taille d'une variable `int` est inférieure ou égale à celle d'une variable `long`.
- La taille d'une variable non signée est égale à la taille d 'une variable `int`.
- La taille d'une variable `float` est inférieure ou égale à la taille d 'une variable `double`.

Les déclarations de variables

Avant d'utiliser une variable dans un programme C, il faut la déclarer. Cette *déclaration* indiquera au compilateur le nom et le type de la variable et elle pourra l'initialiser à une certaine valeur. Si votre programme utilise une variable qui n'a pas été déclarée, le compilateur génère un message d'erreur. Une déclaration de variable a la forme suivante :

```
typename varname;
```

typename indique le type de variable et doit faire partie des mots clés répertoriés dans le Tableau 3.2. *varname* est le nom de la variable, et doit suivre les règles mentionnées plus haut. Vous pouvez déclarer plusieurs variables du même type sur une seule ligne en les séparant par des virgules.

```
int count, number, start;  /* trois variables entières */
float percent, total;      /* deux variables à virgule flottante */
```

Le Chapitre 12 vous apprendra que l'emplacement des déclarations de variable dans le code source est important, parce qu'il affecte la façon dont le programme va utiliser ces variables. À ce stade de votre étude, vous pouvez placer toutes les déclarations de variable juste avant le début de la fonction `main()`.

Le mot clé *typedef*

Le mot clé `typedef` permet de créer un synonyme pour un type de donnée existant. Par exemple, l'instruction :

```
typedef int entier;
```

crée le synonyme `entier` pour `int`. Vous pourrez ainsi utiliser `entier` pour définir des variables de type `int`, comme dans l'exemple suivant :

```
entier compte;
```

`typedef` ne créent pas un nouveau type de donnée, il permet seulement d'utiliser un nom différent pour un type de donnée déjà définie. L'usage le plus fréquent de `typedef` concerne les *données agrégées* qui sont expliquées au Chapitre 11.

Initialisation des variables numériques

La déclaration de variable permet au compilateur de réserver l'espace mémoire destiné à cette variable. La donnée qui sera stockée dans cet emplacement, la valeur de la variable, n'est pas encore définie. Avant d'être utilisée, la variable déclarée doit être initialisée. Cela peut se faire en utilisant une instruction d'initialisation comme dans notre exemple :

```
int count;    /* Réservation de la mémoire pour count */
count = 0;    /* Stocke 0 dans count */
```

Le signe égal fait partie des opérateurs du langage C. En programmation, ce signe n'a pas le même sens qu'en algèbre. Si vous écrivez :

```
x = 12
```

dans une instruction algébrique, vous énoncez un fait : "x = 12". En langage C, la signification est différente : "donner la valeur 12 à la variable appelée x".

Vous pouvez initialiser une variable au moment de sa déclaration. Il suffit de faire suivre le nom de variable, dans l'instruction de déclaration, du signe égal suivi de la valeur initiale :

```
int count = 0;
double percent = 0.01, taxrate = 28.5;
```

Attention, il ne faut pas initialiser la variable avec une valeur qui ne correspond pas au type déclaré. Voici deux exemples d'initialisations incorrectes :

```
int poids = 100000;
unsigned int valeur = -2500;
```

Le compilateur ne détecte pas ce genre d'erreur, et le programme pourrait vous donner des résultats surprenants.



À faire

Connaître la taille en octets des différents types de variables sur votre ordinateur.

Utiliser typedef pour faciliter la lecture de vos programmes.

Initialiser les variables dans l'instruction de déclaration chaque fois que c'est possible.

À ne pas faire

Ne pas utiliser une variable float ou double si vous ne stockez que des valeurs entières.

Ne pas essayer de stocker des nombres dans des variables de type trop petit pour les recevoir.

Ne pas stocker des nombres négatifs dans des variables de type unsigned.

Les constantes

Une *constante* est un emplacement mémoire utilisé par votre programme. À l'inverse d'une variable, la valeur stockée dans une constante ne peut changer pendant l'exécution du programme. Le langage C possède deux types de constantes qui ont chacune un usage spécifique.

Les constantes littérales

Une *constante littérale* est une valeur qui est introduite directement dans le code source. Voici deux exemples :

```
int count = 20;
float tax_rate = 0.28;
```

`20` et `0.28` sont des constantes littérales. Ces deux instructions stockent ces valeurs dans les variables `count` et `tax rate`. La valeur qui contient un point décimal est une constante à virgule flottante, l'autre est une constante entière.

Une constante avec virgule flottante est considérée par le compilateur C comme un nombre double précision. Les constantes avec virgule flottante peuvent être représentées avec une notation décimale standard :

```
123.456
0.019
100.
```

La troisième constante, `100.`, sera traitée par le compilateur C en valeur double précision à cause de son point décimal. Sans point décimal, elle aurait été traitée comme une constante entière.

Les constantes à virgule flottante peuvent être représentées en *notation scientifique*. La notation scientifique représente un nombre par sa partie décimal multipliée par dix à une puissance positive ou négative. Cette notation est particulièrement utile pour exprimer des valeurs très grandes ou très petites. En langage C, le nombre décimal est immédiatement suivi de E ou e puis de l'exposant :

```
1.23E2     1.23 fois 10 à la puissance 2, ou 123
4.08e6     4.08 fois 10 à la puissance 6, ou 4 080 000
0.85e-4    0.85 fois 10 à la puissance -4, ou 0.000085
```

Une constante sans point décimal est considérée comme un nombre entier par le compilateur. Il existe trois notations différentes pour les constantes entières :

- Une constante qui commence par un chiffre différent de 0 est interprétée comme un entier décimal (système numérique standard en base 10). Les constantes décimales s'expriment à l'aide des chiffres 0 à 9 accompagnés d'un signe moins ou plus.
- Une constante qui commence par le chiffre 0 s'exprime en octal (système numérique en base 8). Une constante en octal peut contenir les chiffres 0 à 7 accompagnés du signe moins ou plus.
- Une constante qui commence par 0x ou 0X est interprétée comme une constante hexadécimale (système numérique en base 16). Les constantes hexadécimales s'expriment à l'aide des chiffres 0 à 9, des lettres A à F, et du signe moins ou plus.



Les notations hexadécimales et décimales sont traitées dans l'Annexe C.

Les constantes symboliques

Une constante symbolique est une constante représentée par un nom (symbole) dans votre programme. Comme la constante littérale, cette constante symbolique ne peut changer. Vous utilisez son nom dans le programme chaque fois que vous avez besoin de sa valeur. Cette valeur doit être initialisée une fois au moment de la définition de la variable.

Ces constantes ont deux avantages sur les constantes littérales. Supposons que vous écriviez un programme qui réalise des calculs géométriques. Ce programme aura souvent

besoin de la valeur π (3,14159). Par exemple, pour calculer la circonférence et l'aire d'un cercle dont on connaît le rayon, vous pourriez écrire :

```
circonference = 3.14 159* (2 * rayon);
aire = 3.14 159* (rayon) * (rayon);
```

L'astérisque (`*`) est l'opérateur de multiplication du langage C (voir Chapitre 4). La première instruction signifie "multiplier par 2 la valeur stockée dans la variable `rayon`, puis multiplier le résultat par 3,14159, enfin, stocker le résultat dans la variable `circonférence`".

Si vous définissez une constante symbolique de nom `PI` et de valeur 3,14, vous pourriez écrire :

```
circonference = PI * (2 * rayon);
aire = PI * (rayon) * (rayon);
```

Ces instructions sont plus faciles à lire et à comprendre.

Le second avantage des constantes symboliques apparaît quand vous avez besoin de changer cette constante. Si vous décidez, dans l'exemple précédent, d'utiliser une valeur de p plus précise (3,14159 plutôt que 3,14), vous ne devez changer cette valeur qu'une fois, au niveau de la définition. Avec une constante littérale, vous devez changer chaque occurrence du code source.

Il y a deux méthodes en langage C pour définir une constante symbolique : l'ordre `#define` et le mot clé `const`. `#define` est une commande du préprocesseur qui sera traitée au Chapitre 21.

L'instruction suivante crée une constante appelée `CONSTNAME` avec la valeur `literal` :

```
#define CONSTNAME literal
```

`Literal` représente une constante littérale. Par convention, le nom des constantes symboliques s'écrit en lettres majuscules. Cela permet de les distinguer des noms de variables qui sont par convention en lettres minuscules. Dans l'exemple précédent, la commande `#define` aurait été :

```
#define PI 3.14159
```

Remarquez que cette instruction ne se termine pas par un point-virgule (`;`). La commande `#define` peut se trouver n'importe où dans le code source, mais son effet est limité à la partie de code qui la suit. En général, les programmeurs groupent tous les `#define` ensemble, au début du fichier, avant la fonction `main()`.

Fonctionnement de *#define*

Le rôle de `#define` est d'indiquer au compilateur la directive : "dans le code source, remplacer `CONSTNAME` par `literal`". Vous auriez obtenu le même résultat en faisant tous les changements manuellement avec un éditeur. Bien sur, `#define` ne remplace pas les occurrences qui pourraient se trouver à l'intérieur d'un mot plus long, entre guillemets, ou dans un commentaire du programme. Dans le code suivant, les valeurs de π des deuxième et troisième lignes resteraient identiques :

```
#define PI 3.14159
/* vous avez défini la constante PI. */
#define PIPETTE 100
```

Définition des constantes avec le mot clé *const*

La seconde méthode pour définir une constante symbolique est d 'utiliser le mot clé `const` qui peut modifier n'importe quelle déclaration de variable. Une variable définie avec ce mot clé ne peut être modifiée pendant l'exécution du programme. Voici quelques exemples :

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

`const` s'applique sur toutes les variables de la ligne de déclaration. `debt` et `tax rate` sont des constantes symboliques. Si votre programme essaie de modifier une variable `const`, le compilateur génère un message d'erreur comme dans l'exemple suivant :

```
const int count = 100;
count = 200; /* Pas de compilation ! On ne peut pas changer */
             /* la valeurd'une constante. */
```

Les différences entre une constante symbolique créée avec l'instruction `#define` et une autre, créée avec le mot clé `const`, concernent les pointeurs et la portée des variables. Ce sont deux aspects importants de la programmation C qui sont traités aux Chapitres 9 et 12.

Étudions le code du Listing 3.2 qui illustre ces déclarations de variables et qui utilise des constantes symboliques et littérales. Ce programme demande à l'utilisateur d'entrer son poids et son année de naissance. Il affiche ensuite le poids de l'utilisateur en grammes et l'âge qu'il avait en l'an 2000. Vous pouvez saisir, compiler, et exécuter ce programme en suivant la procédure du Chapitre 1.

Listing 3.2 : Utilisation des variables et des constantes

```
/* Exemple d'utilisation de variables et de constantes */
#include <stdio.h>
#include <stdlib.h>
```

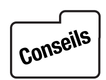
```
 4:  /* Définition d'une constante pour convertir les livres en grammes */
 5:  #define GRAMS_PAR_LIVRE 454
 6:
 7:  /* Définition d'une constante pour le début du siècle */
 8:  const int DEBUT_SIECLE = 2000;
 9:
10:  /* Déclaration des variables requises */
11:  int poids_en_grams, poids_en_livres;
12:  int an_naissance, age_en_2000;
13:
14:  int main()
15:  {
16:         /* Lecture des données de l'utilisateur */
17:
18:         printf("Entrez votre poids en livres : ");
19:         scanf("%d", &poids_en_livres);
20:         printf("Entrez votre année de naissance : ");
21:         scanf("%d", &an_naissance);
22:
23:         /* conversions */
24:
25:         poids_en_grams = poids_en_livres * GRAMS_PAR_LIVRE;
26:         age_en_2000 = DEBUT_SIECLE - an_naissance;
27:
28:         /* Affichage des résultats */
29:
30:         printf("\nVotre poids en grammes = %d", poids_en_grams);
31:         printf("\nEn l'an %d vous avez eu %d ans.\n",
32:         DEBUT_SIECLE,age_en_2000);
33:
34:         exit(EXIT_SUCCESS);
35:  }
```

```
Entrez votre poids en livres : 175
Entrez votre année de naissance : 1990

Votre poids en grammes = 79450
En l'an 2000 vous avez eu 10 ans.
```

Analyse

La déclaration des deux types de constantes symboliques se fait en lignes 5 et 8. La constante de la ligne 5 permet de comprendre facilement la ligne 25. Les lignes 11 et 12 déclarent les variables utilisées dans le programme. Les lignes 18 et 20 demandent à l'utilisateur d'entrer ses données, et les lignes 19 et 21 récupèrent les informations de l'utilisateur à partir de l'écran. Les fonctions de bibliothèque `printf()` et `scanf()` seront étudiées dans les prochains chapitres. Le calcul du poids et de l'âge s'effectue aux lignes 25 et 26, et le résultat est affiché avec les lignes 30 et 31.



À faire

Utiliser des constantes pour faciliter la lecture de votre programme.

À ne pas faire

Essayer de stocker une valeur dans une constante qui a déjà été initialisée.

Résumé

Vous venez d'étudier les variables numériques qui sont utilisées par les programmes C pour stocker des données pendant l'exécution. Il existe deux classes de variables numériques, entière et à virgule flottante, qui ont chacune leurs propres types de variables. Le type que vous choisirez (`int`, `long`, `float`, ou `double`) dépend de la nature de la donnée à stocker dans cette variable. La déclaration doit précéder l'utilisation de cette variable et elle transmet au compilateur le nom et le type de la variable.

À l'inverse des variables, les deux types de constantes, littérale et symbolique, ont une valeur qui ne peut changer pendant l'exécution du programme. La constante littérale est introduite dans le code source au moment de son utilisation. La constante symbolique est créée avec l'instruction `#define` ou avec le mot clé `const`. Elle est référencée par son nom.

Q & R

Q Pourquoi ne pas toujours utiliser les variables `long int` qui peuvent contenir de grands nombres plutôt que des variables `int` ?

R Une variable `long int` peut être plus gourmande en mémoire. Cela ne fait pas de différence dans un petit programme, mais plus il sera gros, plus il deviendra important de bien gérer la mémoire utilisée.

Q Que se passera-t-il si j'essaye de stocker un nombre décimal dans un entier ?

R Vous pouvez stocker un nombre avec une décimale dans une variable `int`. Si cette variable est une variable constante, votre compilateur va certainement vous envoyer un warning. La valeur stockée aura perdu sa partie décimale. Par exemple, si vous donnez la valeur 3,14 à la variable entière pi, pi ne contiendra que la valeur 3.

Q Que se passera-t-il si j'essaye de stocker un nombre dans un type trop petit pour le recevoir ?

R Beaucoup de compilateurs ne signalent pas ce type d'erreur. Le nombre sera tronqué. Par exemple, si vous voulez stocker 32768 dans un entier signé à 2 octets, l'entier

contiendra la valeur –32768. Si vous assignez la valeur 65535 à cet entier, il contiendra la valeur – 1. Si vous soustrayez la valeur maximum sauvegardée dans le Champ vous obtenez la valeur qui sera stockée.

Q Que se passera-t-il si je mets un nombre négatif dans une variable non signée ?

R Comme pour la question précédente, il est possible que votre compilateur ne signale pas ce type d'erreur. Il fera la même transformation qu'avec un nombre trop long. Par exemple, si vous stockez – 1 dans une variable `short unsigned` de 2 octets, le compilateur stockera dans la variable le nombre le plus grand possible (65535).

Q Quelles différences y a-t-il entre une constante symbolique créée avec l'ordre `#define` et une autre créée avec le mot clé `const` ?

R Les différences se situent au niveau des pointeurs et de la portée de la variable. Ces deux aspects importants de la programmation C sont traités aux Chapitres 9 et 12. Retenez aujourd'hui que l'utilisation de #define pour créer des constantes simplifie la lecture de votre programme.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre. Essayez de comprendre les réponses fournies dans l'Annexe G avant de passer au chapitre suivant.

Quiz

1. Quelle est la différence entre une variable entière et une variable à virgule flottante ?
2. Donnez deux raisons d'utiliser une variable à virgule flottante double précision plutôt que la même variable simple précision.
3. Quelles sont les cinq règles de la norme ANSI concernant l'allocation de la taille des variables ?
4. Quelles sont les deux avantages à utiliser une constante symbolique plutôt qu'une constante littérale ?
5. Trouvez deux méthodes pour définir une constante symbolique appelée MAXIMUM qui aurait une valeur de 100.
6. Quels sont les caractères autorisés dans le nom d'une variable C ?

7. Quelles sont les règles à suivre pour créer des noms de variables et de constantes ?
8. Quelle différence y a-t-il entre une constante symbolique et une constante littérale ?
9. Quelle est la valeur minimum que peut prendre une variable de type `short` ?

Exercices

1. Quel type de variable convient le mieux pour stocker les valeurs suivantes :
   a) L'âge d'une personne.
   b) Le poids d'une personne.
   c) Le rayon d'un cercle.
   d) Votre salaire annuel.
   e) Le prix d'un article.
   f) La note la plus haute d'un test (supposons que ce soit toujours 100).
   g) La température.
   h) Le gain d'une personne.
   i) La distance d'une étoile en kilomètres.
2. Donnez un nom approprié à chaque variable de l'exercice 1.
3. Écrivez les déclarations pour les variables de l'exercice 2.
4. Dans la liste suivante, quels sont les noms de variable corrects ?
   a) 123variable.
   b) x.
   c) score_total.
   d) Poids_en_#s.
   e) one.0.
   f) gross-cost.
   g) RAYON.
   h) Rayon.
   i) rayon.
   j) cela_est_une_variable_pour_stocker_la_largeur

4

Instructions, expressions et opérateurs

Les programmes C sont constitués d'instructions qui contiennent, pour la plupart, des expressions et des opérateurs.

Aujourd'hui vous allez étudier :

- Les instructions
- Les expressions
- Les opérateurs logiques, de comparaison et mathématiques du langage C
- Les ordres de priorité ou la hiérarchie des opérateurs
- L'instruction `if`

Les instructions

Une *instruction* représente une tâche à accomplir par l'ordinateur. En langage C, on écrit une instruction par ligne et elle se termine par un point-virgule (à l'exception de `#define` et `#include` qui sont traitées au Chapitre 21). Par exemple :

```
x = 2 + 3;
```

est une *instruction d'affectation*. Elle demande à l'ordinateur d'ajouter 2 et 3 et d'attribuer le résultat à la variable x.

Instructions et blancs

Le terme de *blancs* fait référence à tout espace, tabulation ou ligne de blancs du code source. Quand le compilateur lit une instruction, il traite les caractères et le point-virgule de fin. Il ignore absolument tous les blancs. Par exemple, l'instruction :

```
x=2+3;
```

est équivalente à :

```
x = 2 + 3;
```

ou même à :

```
x   =
2
+
3;
```

Cela vous laisse une grande liberté pour la mise en page de votre code.

Cette règle comporte cependant une exception, les constantes *chaîne de caractères*. Une chaîne est constituée de toute séquence de caractères (y compris les blancs et tabulations) cernée par des guillemets. Le compilateur interprétera la séquence entière. Vous pouvez écrire par exemple :

```
printf(
"Hello, world!"
);
```

La forme n'est pas à suivre, mais la syntaxe est correcte. L'instruction suivante, au contraire, est incorrecte :

```
printf("Hello,
world !");
```

En utilisant l'antislash (\) comme dans l'exemple suivant, vous effectuez un retour à la ligne visible aussi bien dans le code qu'à l'exécution :

```
printf("Hello,\
world !");
```

Un blanc pouvant aisément se cacher après un antislash, préférez l'utilisation de la séquence `\n` pour vos retours à la ligne.

Les instructions nulles

Si vous placez un point-virgule seul sur une ligne, vous avez créé une *instruction nulle*. Cette instruction n'effectue aucune opération, mais vous apprendrez, dans les prochains chapitres, qu'elle peut se révéler utile.

Les blocs

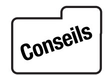
Un *bloc* (ou *instructions composées)* est un groupe d'instructions entre accolades :

```
{
    printf("Hello,");
    printf("world!");
}
```

Les accolades peuvent se positionner de différentes façons. L'exemple suivant est équivalent au précédent :

```
{printf("Hello,");
printf("world!");}
```

En plaçant les accolades sur une ligne séparée, vous identifierez plus facilement le début et la fin du bloc, et vous éviterez d'en oublier une.



À faire

Utiliser les blancs de manière cohérente.

Isoler les accolades, le code sera plus facile à lire.

À ne pas faire

Répartir une instruction sur plusieurs lignes alors que ce n'est pas nécessaire. Il est préférable de respecter la règle d'une instruction par ligne.

Les expressions

En langage C, on appelle *expression* tout ce qui représente une valeur numérique.

Les expressions simples

L'expression la plus simple est constituée d'une seule variable, d'une constante littérale ou d'une constante symbolique. Voici quatre exemples d'expressions :

Expression	***Description***
`PI`	Constante symbolique (définie dans le programme)
`20`	Constante littérale
`taux`	Variable
`1.25`	Constante littérale

La valeur d'une constante littérale est sa propre valeur. La valeur d'une constante symbolique est celle qui a été définie au niveau de l'instruction `#define`. La valeur courante d'une variable est celle qui lui a été attribuée par le programme.

Les expressions complexes

Les expressions complexes sont constituées d'expressions plus simples avec des opérateurs. Par exemple :

```
2 + 8
```

est une expression formée de deux sous-expressions 2 et 8 et de l'opérateur d'addition (+). La valeur de cette expression est 10. Vous pouvez aussi écrire des expressions beaucoup plus complexes :

```
1.25 / 8 + 5 * taux + taux * taux / cout
```

Quand une expression contient plusieurs opérateurs, son évaluation dépend de l'ordre dans lequel les opérations sont effectuées. Ce concept de hiérarchie est expliqué plus loin dans le chapitre.

L'instruction `x = a + 10 ;` calcule l'expression `a + 10` et attribue le résultat à `x`. Comme l'illustre la Figure 4.1, l'instruction entière est elle-même une expression qui attribue à la variable située à gauche du signe égal le résultat du calcul de droite.

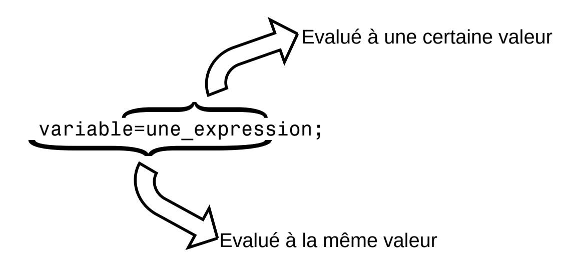


Figure 4.1
Une instruction d'affectation est elle-même une expression.

Ainsi, vous pouvez écrire des instructions comme l'exemple qui suit :

```
y = x = a + b;
```

ou

```
x = 6 +  (y = 4 + 5);
```

L'instruction précédente attribue la valeur `9` à `y`, puis la valeur `15` à `x`.

Les opérateurs

Un *opérateur* est un symbole qui décrit une opération ou une action à effectuer sur une ou plusieurs *opérandes*. En langage C, les opérandes sont toujours des expressions. Les opérateurs sont divisés en quatre catégories :

- l'opérateur d'affectation ;
- les opérateurs mathématiques ;
- les opérateurs de comparaison ;
- les opérateurs logiques.

L'opérateur d'affectation

L'opérateur d'affectation est le signe égale (=). Dans un programme C, l'instruction :

```
x = y;
```

ne signifie pas "x égale y". Elle indique à l'ordinateur "d'affecter la valeur de y à x". Cette instruction doit être composée d'une *expression* à droite du signe égale, et d'un nom de *variable* à gauche de ce signe :

```
variable = expression;
```

Les opérateurs mathématiques

Les opérateurs mathématiques de C réalisent des opérations mathématiques comme l'addition ou la soustraction. Il en existe deux unaires et cinq binaires.

Les opérateurs mathématiques unaires

Les opérateurs unaires opèrent sur une seule valeur ou opérande.

Tableau 4.1 : Les opérateurs mathématiques unaires du langage C

Opérateur	***Symbole***	***Opération***	***Exemples***
Incrémentation	`++`	Augmente de 1 la valeur de l'opérande	`++x, x++`
Décrémentation		Décrémente de 1 la valeur de l'opérande	`x, x`

Ces deux opérateurs ne peuvent être utilisés qu'avec des variables. L'opération réalisée est d'ajouter ou de soustraire 1 de l'opérande. Les instructions :

```
++x;
--y;
```

sont équivalentes aux instructions suivantes :

```
x = x + 1;
y = y - 1;
```

L'opérateur unaire peut être placé avant (*mode préfix*) ou après (*mode postfix*) l'opérande :

- En mode préfix, l'incrémentation et la décrémentation sont effectuées avant l'utilisation de l'opérande.
- En mode postfix les opérateurs d'incrémentation et de décrémentation modifient l'opérande après son utilisation.

Cette explication sera plus claire avec un exemple :

```
x = 10;
y = x++;
```

À la suite de ces deux instructions, `x` a la valeur 11 et `y` la valeur 10. La valeur de `x` a été attribuée à `y`, puis `x` a été incrémenté. Les instructions suivantes, au contraire, donnent à `x` et à `y` la même valeur 11 : `x` est incrémenté, puis sa valeur est affectée à `y`.

```
x = 10;
y = ++x;
```

Souvenez-vous que le signe (=) est l'opérateur d'affectation, et non une instruction d'égalité. Considérez-le comme un opérateur de "photocopie". L'instruction `y = x` signifie "copier `x` dans `y`". Les transformations que pourra ensuite subir `x` n'auront aucun effet sur `y`.

Le Listing 4.1 illustre les différences entre les modes préfix et postfix.

Listing 4.1 : unaire.c

```
 1: /* Démonstration des modes préfix et postfix */
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4:
 5: int a, b;
 6:
 7: int main()
 8: {
 9:     /* initialise a et b à la valeur 5 */
10:
11:    a = b = 5;
12:
13:    /* on les affiche, en les décrémentant chaque fois */
14:    /* mode préfixe pour b, mode postfix pour a */
15:
16:    printf("\n%d    %d", a --, --b);
17:    printf("\n%d    %d", a --, --b);
18:    printf("\n%d    %d", a --, -- b);
19:    printf("\n%d    %d", a --, -- b);
20:    printf("\n%d    %d\n", a --, -- b);
21:
22:    exit(EXIT_SUCCESS);
23: }
```

```
5   4
4   3
3   2
2   1
1   0
```

Analyse

Ce programme déclare les deux variables `a` et `b` en ligne 5 puis leur donne la valeur 5 en ligne 11. Chacune des instructions `printf()` des lignes 16 à 20 décrémente ces variables de 1 : la décrémentation de `a` s'effectue après son affichage, celle de `b` s'effectue avant.

Opérateurs mathématiques binaires

Les opérateurs binaires du langage C travaillent avec deux opérandes.

Tableau 4.2 : Les opérateurs mathématiques binaires du langage C

Opérateur	*Symbole*	*Opération*	*Exemple*
Addition	+	Additionne les deux opérandes	x + y
Soustraction		Soustrait la valeur du second opérande à la valeur du premier	x y
Multiplication	*	Multiplie les deux opérandes	x * y
Division	/	Divise le premier opérande par le second	x / y
Modulo	%	Donne le reste de la division du premier opérande par le second	x % y

Les quatre premiers opérateurs répertoriés dans le tableau sont des opérateurs familiers. *Modulo*, que vous ne connaissez peut-être pas, vous donne le reste de la division du premier opérande par le second. Par exemple, 11 modulo 4 donne la valeur 3.

Le Listing 4.2 vous montre comment utiliser l'opérateur modulo pour convertir des secondes en heures, minutes et secondes.

Listing 4.2 : Utilisation de l'opérateur modulo

```
/* Utilisation de l'opérateur modulo */
/* ce programme converti le nombre de secondes que vous lui */
/* donnerez en heures, minutes, secondes. */
#include <stdio.h>
#include <stdlib.h>

/* Définition des constantes */

#define SECS_PER_MIN 60
#define SECS_PER_HOUR 3600

unsigned seconds, minutes, hours, secs_left, mins_left;

int main()
{
   /* Saisie du nombre de secondes */

   printf("Entrez le nombre de secondes (< 65 000): ");
   scanf("%d", &seconds);

   hours = seconds / SECS_PER_HOUR;
   minutes = seconds / SECS_PER_MIN;
   mins_left = minutes % SECS_PER_MIN;
```

```
24:    secs_left = seconds % SECS_PER_MIN;
25:
26:   printf("%u secondes représentent ", seconds);
27:   printf("%u h, %u m, et %u s\n", hours, mins_left, secs_left);
28:
29:   exit(EXIT_SUCCESS);
30: }
```

```
$ list4_2
Entrez le nombre de secondes (< 65 000) : 60
60 secondes correspondent à 0 h, 1 m, et 0 s

$ list4_2
Entrez le nombre de secondes (< 65 000) : 10000
10000 secondes correspondent à 2 h, 46 m, et 40 s
```

Analyse

Les commentaires des lignes 1 à 3 indiquent ce que fait le programme. La ligne 5 appelle l'indispensable fichier en-tête et les lignes 8 et 9 définissent les deux constantes `SECS PER MIN` et `SECS PER HOUR`. Les déclarations de variables se font en ligne 12. Certains programmeurs préfèrent déclarer une variable par ligne. Comme avec beaucoup d'éléments du langage C, vous pouvez choisir le style que vous voulez.

La fonction principale `main()` se trouve en ligne 14. La ligne 18, avec la fonction `printf()`, demande à l'utilisateur de taper le nombre de secondes qui est récupéré par le programme avec la fonction `scanf()` de la ligne 19. Cette fonction stocke la valeur lue dans la variable `seconds`. Le Chapitre 7 vous donnera plus de détails sur les deux fonctions `printf()` et `scanf()`. La ligne 21 est une expression qui calcule le nombre d'heures en divisant le nombre de secondes par la constante `SECS PER HOUR`. `hours` étant une variable entière, la valeur restante est ignorée. La ligne 22 utilise la même logique pour déterminer le nombre total de minutes. Les lignes 23 et 24 utilisent l'opérateur modulo pour diviser respectivement les heures et les minutes et conserver les minutes et les secondes restantes. Les lignes 26 et 27 affichent les valeurs calculées. Ce programme se termine en renvoyant la valeur 0 en ligne 29.

Hiérarchie des opérateurs et parenthèses

Quand une expression possède plusieurs opérateurs, l'ordre dans lequel les opérations sont effectuées est important :

```
x = 4 + 5 * 3;
```

Si la première opération réalisée est l'addition, cela revient à l'instruction suivante et x prend la valeur 27 :

```
x = 9 * 3;
```

Au contraire, si la première opération est la multiplication, vous obtenez l'instruction qui suit et x prend la valeur 19 :

```
x = 4 + 15;
```

Des règles de priorité, appelées *hiérarchie des opérateurs*, sont nécessaires. Le Tableau 4.3 vous présente la hiérarchie des opérateurs mathématiques du langage C en commençant par le plus "prioritaire".

Tableau 4.3 : Hiérarchie des opérateurs mathématiques du langage C

Opérateurs	***Priorité d'exécution***
`++`	1
`* / %`	2
`+`	3

Si une expression contient plusieurs opérateurs de même niveau de priorité, les opérations sont réalisées de gauche à droite.

Dans l'exemple qui suit, l'opérateur modulo se trouvant à gauche, (`12 % 5`) sera la première opération effectuée :

```
12 % 5 * 2
```

La valeur de cette expression est 4 (`12 % 5` donne 2; 2 fois 2 donne 4).

Pour modifier l'ordre de validation des opérations, le langage C permet d'utiliser des parenthèses. La sous-expression entre parenthèses est la première calculée quelle que soit la hiérarchie des opérateurs. Dans le cas de notre premier exemple, si vous voulez ajouter 4 et 5 avant de les multiplier par trois, vous pourriez écrire :

```
x = (4 + 5) * 3;
```

La valeur attribuée à `x` est 27.

Une expression peut contenir des parenthèses multiples ou imbriquées. Dans le cas de parenthèses imbriquées, l'évaluation se fait de "l'intérieur" vers "l'extérieur". L'expression :

```
x = 25 - (2 * (10 +  (8 / 2)));
```

se calcule dans l'ordre suivant :

1. `25 - (2 * (10 + 4))`
2. `25 - (2 * 14)`

3. `25 - 28`
4. L'expression finale `x =  3` attribue la valeur –3 à `x`.

Vous pouvez placer des parenthèses pour rendre une expression plus facile à comprendre. Elles doivent toujours aller par paires, sinon le compilateur génère un message d'erreur.

Ordre de traitement des sous-expressions

Une expression qui contient plusieurs opérateurs de même niveau de priorité est évaluée de gauche à droite. Dans l'expression :

```
w * x / y * z
```

`w` est multiplié par `x`, le résultat de la multiplication est divisé par `y` et le résultat de la division est multiplié par `z`.

Si l'expression contient de multiples opérateurs de priorités différentes, l'ordre de traitement de gauche à droite n'est plus garanti. Étudions l'exemple suivant :

```
w * x / y + z / y
```

La multiplication et la division doivent être traitées avant l'addition. Les règles du langage ne permettent pas de savoir si `w * x / y` doit être calculé avant ou après `z / y`. Si on transforme notre expression, le résultat sera différent selon l'ordre dans lequel seront évaluées les sous-expressions :

```
w * x / ++y + z / y
```

Si la sous-expression de gauche est la première calculée, `y` est incrémenté quand la seconde expression est évaluée. Si le calcul commence avec l'expression de droite, `y` n'est pas incrémenté et le résultat est différent. Vous devez éviter d'utiliser ce genre d'expression indéterminée.

La hiérarchie de tous les opérateurs du langage C vous sera fournie à la fin de ce chapitre.



À faire

Utiliser des parenthèses pour que l'ordre d'évaluation des expressions ne soit pas ambigu.

À ne pas faire

Surcharger une expression. Elle devient souvent plus claire si on la divise en plusieurs sous-expressions, tout particulièrement avec des opérateurs unaires (—) ou (++).

Les opérateurs de comparaison

Les *opérateurs de comparaison* sont utilisés pour comparer des expressions en posant des questions du type "x est-il plus grand que 100 ?" ou "y est-il égal à 0 ?". La valeur finale d'une expression qui contient un opérateur de comparaison est "vrai" (différent de `0`) ou "faux" (`0`).



"Vrai" est équivalent à 1 ou "oui". "Faux" est équivalent à 0 ou "non".

Tableau 4.4 : Les opérateurs de comparaisons du langage C

Opérateur	***Symbole***	***Question posée***	***Exemple***
Égal	`==`	Le premier opérande est-il égal au second ?	`x == y`
Supérieur	`>`	Le premier opérande est-il plus grand que le second ?	`x > y`
Inférieur	`<`	Le premier opérande est-il plus petit que le second ?	`x < y`
Supérieur ou égal	`>=`	Le premier opérande est-il supérieur ou égal au second ?	`x >= y`
Inférieur ou égal	`<=`	Le premier opérande est-il inférieur ou égal au second ?	`x <= y`
Différent	`!=`	Le premier opérande est-il différent du second ?	`x != y`

Tableau 4.5 : Exemples d'utilisations des opérateurs de comparaison

Expression	***Signification***	***Valeur***
`5 == 1`	La valeur 5 est-elle égale à 1 ?	`faux`
`5 > 1`	5 est-elle plus grande que 1 ?	`vrai`
`5 != 1`	La valeur 5 est-elle différente de 1 ?	`vrai`
`(5 + 10) == (3 * 5)`	L'expression (5 + 10) est-elle égale à (3 * 5) ?	`vrai`



À faire

Comprendre que le langage C interprète une expression vraie comme ayant une valeur non nulle et une expression fausse comme ayant la valeur 0.

À ne pas faire

Confondre l'opérateur de comparaison (==) avec l'opérateur d'affectation (=). C'est une des erreurs les plus courantes des programmeurs.

Utiliser le résultat d'un test (généralement la valeur 1) dans une expression arithmétique.

L'instruction if

Les opérateurs de comparaison sont principalement utilisés dans les instructions `if` et `while` pour le contrôle de l'exécution du programme.

Ce contrôle permet de modifier la règle suivante : les instructions d'un programme C s'exécutent en séquence, dans l'ordre dans lequel elles sont placées dans le fichier source. Une *structure de contrôle* modifie l'ordre d'exécution des instructions. Une instruction de contrôle peut provoquer l'exécution de certaines instructions du programme plusieurs fois, ou pas d'exécution du tout selon les circonstances. L'instruction `if` en fait partie, ainsi que `do` et `while` qui sont traitées dans le Chapitre 6.

L'instruction if évalue une expression, et oriente l'exécution du programme en fonction du résultat de cette évaluation. La syntaxe est la suivante :

```
if (expression)
   instruction;
```

Si le résultat de l'évaluation est vrai, l'instruction est exécutée. Dans le cas contraire, l'exécution du programme se poursuit avec l'instruction qui suit l'instruction `if`. Notez que les deux lignes de notre exemple constituent l'instruction `if`, ce ne sont pas des instructions séparées.

Une instruction `if` peut contrôler l'exécution de nombreuses lignes de code par l'intermédiaire d'un bloc. Comme nous l'avons défini, un bloc est constitué d'un groupe d'instructions cernées par des accolades et il est utilisé de la même façon qu'une instruction. L'instruction `if` peut donc prendre la forme suivante :

```
if (expression)
{
   instruction 1;
   instruction 2;
   /* code supplémentaire si nécessaire */
   instruction n;
}
```



À faire

Décaler les instructions à l'intérieur d'un bloc pour. Cela concerne aussi les instructions `if`*.*

À ne pas faire

Mettre un point-virgule à la fin de l'instruction `if`*. Cette instruction doit se terminer par une instruction de comparaison. Dans l'exemple suivant,* `instruction 1` *est exécutée quel que soit le résultat de la comparaison, car chaque ligne est évaluée comme une instruction indépendante :*

```
if (x == 2);  /* il ne devrait pas y avoir de point-virgule ! */
instruction 1;
```

Au cours de votre programmation, vous vous apercevrez que les instructions `if` sont surtout utilisées avec des expressions de comparaison. En d'autres termes, "Exécute l'instruction suivante seulement si telle condition est vraie". Voici un exemple :

```
if (x > y)
   y = x;
```

`y` prendra la valeur de `x` seulement si `x` est plus grand que `y`. Si `x` est plus petit, l'instruction n'est pas exécutée.

Listing 4.3 : L'instruction if

```
/* Exemple d'utilisation de l'instruction de contrôle if */
#include <stdio.h>
#include <stdlib.h>

int x, y;

int main()
{
/* Lecture des deux valeurs à tester */

   printf("\nEntrez une valeur entière pour x : ");
   scanf("%d", &x);
   printf("\nEntrez une valeur entière pour y : ");
   scanf("%d", &y);

/* Test des valeurs et affichage des résultats */

   if (x == y)
     _ printf("x est égal à y\n");

   if (x > y)
     _ printf("x est plus grand que y\n");

```

```
24:    if (x < y)
25:      _ printf("x est plus petit que y\n");
26:
27:    exit(EXIT_SUCCESS);
28: }
```



```
Entrez une valeur entière pour x : 100

Entrez une valeur entière pour y : 10
x est plus grand que y

Entrez une valeur entière pour x : 10

Entrez une valeur entière pour y : 100
x est plus petit que y

Entrez une valeur entière pour x : 10

Entrez une valeur entière pour y : 10
x est égal à y
```

Analyse

list4_3.c contient trois instructions `if` (lignes 18 à 25). La ligne 5 déclare les deux variables `x` et `y`, et les lignes 11 à 14 demandent à l'utilisateur d'entrer des valeurs pour ces variables. Les lignes 18 à 25 utilisent l'instruction `if` pour savoir si `x` est égal, supérieur ou inférieur à `y` et affichent le résultat.

Info

Les instructions à l'intérieur de l'instruction `if` sont décalées pour faciliter la lecture du programme.

La clause else

Une instruction `if` peut contenir une clause `else` comme le montre l'exemple suivant :

```
if (expression)
    instruction1;
else
   instruction2;
```

Si `expression` est évaluée comme étant vraie, `instruction1` est exécutée, sinon c'est `instruction2` qui est exécutée. Ces deux instructions peuvent être remplacées par des blocs.

Le Listing 4.4 vous présente le Listing 4.3 réécrit avec des clauses `else`.

Listing 4.4 : L'instruction if avec une clause else

```
 1: /*   Exemple d'utilisation de l'instruction if avec la clause else   */
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4:
 5: int x, y;
 6:
 7: int main()
 8: {
 9:      /* Lecture des deux valeurs à tester */
10:
11:     printf("\nEntrez une valeur entière pour x: ");
12:     scanf("%d", &x);
13:     printf("\nEntrez une valeur entière pour y: ");
14:     scanf("%d", &y);
15:
16:     /* Test des valeurs et affichage des résultats */
17:
18:     if (x == y)
19:         printf("x est égal à y\n");
20:     else
21:         if (x > y)
22:             printf("x est plus grand que y\n");
23:         else
24:             printf("x est plus petit que y\n");
25:
26:     exit(EXIT_SUCCESS);
27: }
```

```
Entrez une valeur entière pour x : 99

Entrez une valeur entière pour y : 8
x est plus grand que y

Entrez une valeur entière pour x : 8

Entrez une valeur entière pour y : 99
x est plus petit que y

Entrez une valeur entière pour x : 99

Entrez une valeur entière pour y : 99
x est égal à y
```

Analyse

Les lignes 18 à 24 sont légèrement différentes du code source précédent. La ligne 18 contrôle toujours si `x` est égal à `y`, mais si la comparaison est vraie, "`x est égal à y`" est affiché et le programme se termine sans exécuter les lignes 20 à 24. La ligne 21 n'est

exécutée que si l'expression "x égal y" est fausse. Si x est plus grand que y, la ligne 22 affiche le message sinon la ligne 24 est exécutée.

Ce listing a utilisé une instruction if *imbriquée*. Cela signifie que cette instruction if fait partie d'une autre instruction if. Dans notre exemple, L'instruction imbriquée fait partie de la clause else de la première instruction if.

Syntaxe de la commande *if*

Forme 1

```
if (expression)
   instruction1;
instruction_suivante;
```

L'instruction if est ici dans sa forme la plus simple. Si expression est vraie, instruction1 est exécutée. Si expression est fausse, instruction1 est ignorée.

Forme 2

```
if (expression)
   instruction1;
else
   instruction2;
instruction suivante;
```

C'est la forme la plus courante. Si expression est vraie, instruction1 est exécutée, sinon c'est instruction2 qui est exécutée.

Forme 3

```
if (expression1)
   instruction1;
else if (expression2)
   instruction2;
else
   instruction3;
instruction suivante;
```

Les instructions if sont imbriquées. Si expression1 est vraie, instruction1 est exécutée. Dans le cas contraire, expression2 est évaluée. Si cette dernière est vraie, instruction2 est exécutée. Si les deux expressions sont fausses, c'est instruction3 qui est exécutée.

Exemple 1

```
if (salaire > 45,0000)
    taxe = .30;
else
    taxe = .25;
```

Exemple 2

```
if (age < 18)
    printf("mineur");
else if (age < 65)
    printf("adulte");
else
    printf("personne agée");
```

Évaluation des expressions de comparaison

Une expression de comparaison est évaluée à la valeur (`0`) si elle est fausse, et à une valeur non nulle (généralement `1`) si elle est vraie. Bien que ce type d'expression soit presque toujours inclus dans une structure de contrôle (exemple `if`) sa valeur numérique peut être utilisée. Cela est à proscrire car la valeur vraie peut prendre une valeur autre que 1 dans certaines circonstances.

Listing 4.5 : Évaluation des expressions de comparaison

```
/* Exemple de l'évaluation d'expressions de comparaison */
#include <stdio.h>
#include <stdlib.h>

int a;

int main()
{
     a = (5 == 5);    /* Évalué à priori à 1 */
    printf("a = (5 == 5)\na = %d\n", a);

    a = (5 != 5);    /* Évalué à 0 */
    printf("a = (5 != 5)\na = %d\n", a);

    a = (12 == 12)?1:0 +  (5 != 1)?1:0;    /* Évalué à 1 + 1 */
    printf("\na = (12 == 12)?1:0 +  (5 != 1)?1:0\na = %d\n", a);
    exit(EXIT_SUCCESS);
}
```



```
a = (5 == 5)
a = 1
a = (5 != 5)
a = 0
a = (12 == 12) +  (5 != 1)
a = 2
```

Analyse

Le résultat de l'exécution de ce programme peut vous paraître confus. Rappelez-vous, l'erreur la plus courante avec les opérateurs de comparaison est d'utiliser l'opérateur d'affectation (=) en lieu et place de l'opérateur (==). La valeur de l'expression suivante est 5 (cette valeur est aussi stockée dans x) :

```
x = 5
```

L'expression qui suit, au contraire, est évaluée à vrai ou faux (selon l'égalité de `x` avec 5), mais elle ne change pas la valeur de `x` :

```
x == 5
```

Si vous écrivez :

```
if (x = 5)
    printf("x est égal à 5");
```

le message apparaîtra dans tous les cas, car l'expression sera toujours évaluée comme vraie, quelle que soit la valeur de `x`.

Vous comprenez maintenant pourquoi le programme du Listing 4.5 donne de telles valeurs à `a`. En ligne 9, l'expression `5 == 5` étant toujours vraie, c'est la valeur 1 qui est stockée dans `a`. En ligne 12, l'expression 5 différent de 5 étant fausse, c'est la valeur 0 qui est attribuée à `a`. En ligne 15, l'opérateur de condition (voir plus loin) permet de renvoyer 1 ou 0 en fonction du test.

Hiérarchie des opérateurs de comparaison

Comme pour les opérateurs mathématiques, les opérateurs de comparaison sont traités dans un ordre donné. L'usage des parenthèses permet de modifier l'ordre de sélection dans une instruction qui contient plusieurs opérateurs de comparaison. La hiérarchie de tous les opérateurs du langage C vous sera fournie à la fin de ce chapitre.

Les opérateurs de comparaison ont tous une priorité de traitement inférieure à celle des opérateurs mathématiques. Si vous écrivez :

```
if (x + 2 > y)
```

2 est ajouté à la valeur de `x`, et le résultat est comparé à `y`. L'instruction suivante, équivalente à la première, est un bon exemple d'utilisation des parenthèses pour améliorer la lecture du code :

```
if ((x + 2) > y)
```

Comme nous le montre le Tableau 4.6, il existe aussi deux niveaux de priorité pour le traitement des opérateurs de comparaison.

Tableau 4.6 : Hiérarchie des opérateurs de comparaison du langage C

Opérateur	*Ordre de traitement*
`< <= > >=`	1
`!= ==`	2

L'instruction :

```
x == y > z
```

est équivalente à l'instruction :

```
x == (y > z)
```

car l'expression `y > z` est la première évaluée et son résultat, 0 ou 1, est affecté à la variable `x`.



À ne pas faire

Introduire des instructions d'affectation dans une instruction `if`. Cela peut induire en erreur la personne qui lira le code et elle risque de corriger en (==), en pensant à une faute de programmation.

Utiliser l'opérateur différent (`!=`) dans une instruction `if` qui a une clause `else`. Il est souvent plus clair d'utiliser l'opérateur égal (==) avec cette clause. Par exemple, le code suivant :

```
if (x != 5)
   instruction1;
else
   instruction2;
```

s'écrirait mieux ainsi :

```
if (x == 5)
   instruction2;
else
   instruction1;
```

Les opérateurs logiques

Les opérateurs logiques de C permettent de vérifier plusieurs comparaisons dans une même question. Par exemple "s'il est 7 heures du matin, un jour de semaine, et que je ne suis pas en vacances, faire sonner le réveil".

Tableau 4.7 : Les opérateurs logiques du langage C

Opérateur	*Symbole*	*Exemple*
`ET`	`&&`	`exp1 && exp2`
`OU`	`||`	`exp1 || exp2`
`NON`	`!`	`!exp1`

Tableau 4.8 : Utilisation des opérateurs logiques

Expression	*Valeur*
`(exp1 && exp2)`	Vraie si *exp1* et *exp2* vraies. Sinon faux
`(exp1 || exp2)`	Vraie si *exp1* vraie ou *exp2* vraie. Faux si les deux expressions sont fausses.
`(!exp1)`	Faux si *exp1* est vraie. Vraie si *exp1* est fausse.

Les expressions qui contiennent des opérateurs logiques sont vraies ou fausses selon que leurs opérandes sont eux-mêmes vrais ou faux.

Tableau 4.9 : Exemples d'utilisation des opérateurs logiques

Expression	*Valeur*
`(5 == 5) && (6 != 2)`	Vraie, car les deux opérandes sont vrais
`(5 > 1) || (6 < 1)`	Vraie, car un opérande est vrai
`(2 == 1) && (5 == 5)`	Faux, car un opérande est faux
`!(5 == 4)`	Vrai, car l'opérande est faux

Vous pouvez créer des expressions avec plusieurs opérateurs logiques. Par exemple, la question "x est-il égal à 2, 3, ou 4?" se traduit par :

```
(x == 2) || (x == 3) || (x == 4)
```

Les opérateurs logiques offrent souvent plusieurs solutions pour poser une question. Si `x` est une variable entière, la question précédente pourrait s'écrire des deux façons suivantes :

```
(x > 1) && (x < 5)
(x >= 2) && (x <= 4)
```

Les valeurs VRAI/FAUX

Les expressions de comparaison du C ont la valeur 0 si elles sont fausses et différente de 0 si elles sont vraies. À l'inverse, une valeur numérique sera interprétée en vrai ou faux si elle se trouve dans une expression ou instruction qui attend une valeur logique (c'est-à-dire vrai ou faux). La règle est la suivante :

- Une valeur de 0 signifie faux.
- Une valeur différente de 0 signifie vrai.

Voici un exemple dans lequel `x` sera toujours affiché :

```
x = 125;
if (x)
   printf("%d", x);
```

`x` étant différent de zéro, l'instruction `if` interprète l'expression `(x)` comme vraie. Vous pouvez généraliser cette caractéristique, car, pour une expression C, écrire :

```
(expression)
```

est équivalent à :

```
(expression != 0)
```

Dans les deux cas, le résultat est vrai si `expression` est différent de zéro, et faux si `expression` a la valeur 0. En utilisant l'opérateur non (`!`), vous pouvez écrire aussi :

```
(!expression)
```

qui est équivalent à :

```
(expression == 0)
```

Hiérarchie des opérateurs logiques

Les opérateurs logiques ont aussi leur priorité de traitement, entre eux ou en liaison avec les autres types d'opérateurs. L'opérateur ! a la même priorité que les opérateurs mathématiques unaires ++ et —. Il sera donc traité avant les opérateurs de comparaison et avant les opérateurs mathématiques binaires.

Au contraire, les opérateurs `&&` et `||` seront traités après tous les autres opérateurs mathématiques et de comparaison, `&&` ayant une priorité supérieure à celle de `||`. Comme avec tous les autres opérateurs du langage C, les parenthèses peuvent modifier l'ordre de traitement. Examinons l'exemple suivant.

Vous voulez écrire une expression logique qui effectue trois comparaisons :

1. `a` est-il plus petit que `b` ?
2. `a` est-il plus petit que `c` ?
3. `c` est-il plus petit que `d` ?

Vous voulez une expression logique qui soit vraie si la condition 3 est vraie et si l'une des deux premières conditions est vraie. Vous pourriez écrire :

```
a < b || a < c && c < d
```

Mais cette expression ne donnera pas le résultat escompté. L'opérateur `&&` ayant une priorité supérieure à `||`, l'expression est l'équivalent de :

```
a < b || (a < c && c < d)
```

Si (`a < b`) est vraie, l'expression précédente est vraie quel que soit le résultat de (`a < c`) et (`c < d`). Il faut écrire :

```
(a < b || a < c) && c < d
```

qui force le traitement de `||` avant celui de `&&`. Cela est illustré par le Listing 4.6 qui évalue une expression écrite de deux façons différentes. Les variables sont initialisées pour que l'expression correcte soit égale à 0 (fausse).

Listing 4.6 : Hiérarchie des opérateurs logiques

```
#include <stdio.h>
#include <stdlib.h>
/* Initialisation des variables. Notez que c n'est pas */
/* inférieur à d, ce qui est une des conditions à tester. */
/* L'expression complète doit finalement être fausse. */

int a = 5, b = 6, c = 5, d = 1;
int x;
```

Listing 4.6 : Hiérarchie des opérateurs logiques *(suite)*

```
10: int main()
11: {
12:     /* Évaluation de l'expression sans parenthèses */
13:
14:     x = a < b || a < c && c < d;
15:     printf("Sans parenthèses l'expression a la valeur %d\n", x);
16:
17:     /* Évaluation de l'expression avec parenthèses */
18:
19:     x = (a < b || a < c) && c < d;
20:     printf("Avec les parenthèses l'expression a la valeur %d\n",
                 x);
21:     exit(EXIT_SUCCESS);
22: }
```



```
Sans parenthèses l'expression a la valeur 1
Avec des parenthèses l'expression a la valeur 0
```

Analyse

Ce programme initialise, en ligne 7, les quatre variables qui vont être utilisées dans les comparaisons. La ligne 8 déclare la variable `x` qui sera utilisée pour stocker les résultats. Les opérateurs logiques se trouvent en lignes 14 et 19. La ligne 14 n'utilise pas les parenthèses, le résultat est donc fonction de la hiérarchie des opérateurs et ne correspond pas au résultat recherché. La ligne 19 utilise les parenthèses pour changer l'ordre de traitement des opérateurs.

Les opérateurs d'affectation composés

Ces *opérateurs composés* permettent d'associer une opération mathématique binaire avec une opération d'affectation. Pour, par exemple, augmenter la valeur de `x` de 5, il faut ajouter 5 à `x` et stocker le résultat dans la variable `x` :

```
x = x + 5;
```

L'opérateur composé permet d'écrire :

```
x += 5;
```

Les opérateurs d'affectation composés ont la syntaxe suivante (`op` représente un opérateur binaire) :

```
exp1 op= exp2;
```

qui est l'équivalent de :

```
exp1 = exp1 op exp2;
```

Vous pouvez créer un opérateur composé à partir des cinq opérateurs mathématiques binaires. Le Tableau 4.10 vous donne quelques exemples.

Tableau 4.10 : Exemples d'opérateurs composés

Taper ceci...	*... est équivalent à...*
`x *= y`	`x = x * y`
`y  = z + 1`	`y = y   z + 1`
`a /= b`	`a = a / b`
`x += y / 8`	`x = x + y / 8`
`y %= 3`	`y = y % 3`

Ces opérateurs fournissent une notation raccourcie agréable, surtout si la variable de gauche a un nom très long. Comme pour toutes les autres instructions d'affectation, ce type d'instruction est une expression dont la valeur est affectée à la variable située à gauche. Si vous exécutez l'instruction suivante, les deux variables `x` et `z` auront la valeur 14 :

```
x = 12;
z = x += 2;
```

L'opérateur de condition

L'opérateur de condition est le seul opérateur *ternaire*, ce qui signifie qu'il prend trois opérandes. La syntaxe est la suivante :

```
exp1 ? exp2 : exp3;
```

Si `exp1` est vraie (c'est-à-dire différente de zéro), l'expression complète est évaluée à la valeur de `exp2`. Si `exp1` est fausse, l'expression complète prendra la valeur de `exp3`. L'exemple suivant, par exemple, affecte la valeur 1 à `x` si `y` est vraie, et stocke la valeur 100 dans `x` si `y` est fausse :

```
x = y ? 1 : 100;
```

De la même façon, pour affecter à `z` la valeur de la plus grande variable `x` ou `y`, vous pourriez écrire :

```
z = (x > y) ? x : y;
```

Cet opérateur a le même mode de fonctionnement que l'instruction `if`. L'instruction précédente aurait pu s'écrire de cette façon :

```
if (x > y)
    z = x;
else
    z = y;
```

L'opérateur de condition ne peut pas remplacer toutes les instructions `if`, mais il a l'avantage d'être plus concis. Vous pouvez l'utiliser là ou vous ne pouvez pas utiliser `if`, comme dans une instruction `printf()` :

```
printf( "La valeur la plus élevée est %d", ((x>y) ? x:y) );
```

La virgule

La virgule est souvent utilisée en langage C comme une marque de ponctuation pour séparer les déclarations de variables, les arguments de fonctions, etc.. Dans certains cas, cette virgule agit comme un opérateur. Vous pouvez former une expression en séparant deux sous-expressions par une virgule. Le résultat est le suivant :

- Les deux expressions sont évaluées en commençant par celle de gauche.
- L'expression entière prend la valeur de l'expression de droite.

L'instruction qui suit attribue la valeur de `b` à `x`, incrémente `a`, puis incrémente `b` :

```
x = (a++ , b++);
```

L'opérateur `++` étant utilisé en mode postfix, la valeur de `b` avant son incrémentation est stockée dans `x`. L'usage des parenthèses est obligatoire, car la virgule a une priorité inférieure à celle du signe `=`.



À faire

Utiliser (`expression == 0`) plutôt que (`!expression`). Ces deux expressions ont le même résultat après compilation, mais la première est plus facile à lire.

Utiliser les opérateurs `&&` et `||` plutôt que d'imbriquer des instructions `if`.

À ne pas faire

Confondre l'opérateur d'affectation (=) avec l'opérateur égal à (==).

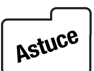
Réorganisation de la hiérarchie des opérateurs

Le Tableau 4.11 établit un classement hiérarchique de tous les opérateurs C dans un ordre décroissant. Les opérateurs placés sur la même ligne se situent au même niveau hiérarchique.

Tableau 4.11 : Hiérarchie des opérateurs C

<table>
<tr><th>Niveau</th><th>Opérateurs</th></tr>
<tr><td>1</td><td><code>() [] > .</code></td></tr>
<tr><td>2</td><td><code>! ~ ++ *</code> (indirection) & (adresse-de) <code>(type)</code></td></tr>
<tr><td></td><td><code>sizeof +</code> (unaire) - (unaire)</td></tr>
<tr><td>3</td><td><code>*</code> (multiplication) <code>/ %</code></td></tr>
<tr><td>4</td><td><code>+</code></td></tr>
<tr><td>5</td><td><code><< >></code></td></tr>
<tr><td>6</td><td><code>< <= > >=</code></td></tr>
<tr><td>7</td><td><code>== !=</code></td></tr>
<tr><td>8</td><td><code>&</code> (ET bit à bit)</td></tr>
<tr><td>9</td><td><code>^</code></td></tr>
<tr><td>10</td><td><code>|</code></td></tr>
<tr><td>11</td><td><code>&&</code></td></tr>
<tr><td>12</td><td><code>||</code></td></tr>
<tr><td>13</td><td><code>?:</code></td></tr>
<tr><td>14</td><td><code>= += = *= /= %= &= ^= |= <<= >>=</code></td></tr>
<tr><td>15</td><td><code>,</code></td></tr>
</table>

`()` représente l'opérateur fonction; `[]` représente l'opérateur tableau.



Ce tableau permettra de vous familiariser avec un ordre hiérarchique qui vous sera indispensable dans l'avenir.

Résumé

Nous avons appris, dans ce chapitre, ce que représente une instruction du langage C, que les blancs de votre programme source sont ignorés par le compilateur, et que le point-virgule doit toujours terminer une instruction. Vous savez maintenant que vous pouvez utiliser un bloc (plusieurs instructions entre accolades) partout où l'on peut utiliser une instruction simple.

Beaucoup d'instructions sont constituées d'expressions et d'opérateurs. Rappelez-vous qu'une expression représente une valeur numérique. Une expression complexe est composée de plusieurs expressions simples appelées sous-expressions.

Les opérateurs sont des symboles du langage C qui indiquent à l'ordinateur d'effectuer une opération sur une ou plusieurs expressions. Il existe des opérateurs unaires (qui agit sur une opérande unique), mais la majorité sont des opérateurs binaires qui opèrent sur deux opérandes. L'opérateur de condition est le seul opérateur ternaire (trois opérandes). Les opérateurs ont une hiérarchie de traitement qui détermine l'ordre dans lequel les opérations d'une expression sont réalisées.

Les opérateurs sont divisés en trois catégories :

- Les opérateurs mathématiques qui effectuent des opérations arithmétiques sur leurs opérandes (l'addition, par exemple).
- Les opérateurs de comparaison qui comparent leurs opérandes (plus grand que, par exemple).
- Les opérateurs logiques qui opèrent sur des expressions vrai/faux. N'oubliez pas que vrai et faux sont représentés par 1 et 0 en langage C, et qu'une valeur différente de zéro est interprétée comme vraie.

Vous avez appris que l'instruction `if` permet de contrôler l'exécution du programme en évaluant des expressions de comparaison.

Q & R

Q Quels sont les effets des blancs et des lignes vides du fichier source sur l'exécution du programme ?

R Tous les blancs (lignes, espaces, tabulations) permettent de rendre le programme source plus facile à lire et à comprendre. Au moment de la compilation, tous ces blancs sont ignorés, ils n'ont donc aucune influence sur l'exécution du programme.

Q Faut-il choisir d'écrire une instruction if composée ou des instructions if imbriquées ?

R Il faut simplifier votre code source. Si vous imbriquez des instructions if, l'évaluation se fait comme nous l'avons vue dans ce chapitre. Si vous utilisez une instruction composée, les expressions ne sont évaluées que si l'expression complète est fausse.

Q Quelle est la différence entre un opérateur unaire et un opérateur binaire ?

R Un opérateur unaire agit sur un seul opérande. Un opérateur binaire opère avec deux opérandes.

Q L'opérateur de soustraction est-il unaire ou binaire ?

R Les deux ! Le compilateur saura lequel vous utilisez en fonction du nombre de variables utilisées dans l'expression. Dans l'instruction suivante il est unaire :

```
x = -y;
```

Dans celle-ci, il est binaire :

```
x = a - b;
```

Q Comment sont évalués les nombres négatifs : en vrai ou faux ?

R Rappelez-vous, 0 est évalué faux, toutes les autres valeurs sont évaluées vraies. Les nombres négatifs en font partie.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre. Essayez de comprendre les réponses fournies dans l'Annexe G avant de passer au chapitre suivant.

Quiz

1. Que fait l'instruction suivante ?

```
x = 5 + 8;
```

2. Qu'est-ce qu'une expression ?
3. Qu'est-ce qui détermine l'ordre de réalisation des opérations dans une expression qui contient plusieurs opérateurs ?

4. Si la variable `x` a la valeur 10, quelles sont les valeurs stockées dans `x` et `a` après l'exécution de chacune de ces instructions (séparément) ?

```
a = x++;
a = ++x;
```

5. Quelle est la valeur de l'expression `10 % 3` ?
6. Quelle est la valeur de l'expression `5 + 3 * 8 / 2 + 2` ?
7. Écrivez l'expression de la question 6 avec des parenthèses pour obtenir le résultat 16.
8. Quelle valeur prend une expression fausse ?
9. Dans la liste suivante, quel opérateur est le plus prioritaire ?
   a) == ou <.
   b) * ou +.
   c) != ou ==.
   d) >= ou >.
10. Qu'est-ce qu'un opérateur d'affectation composé et quel intérêt a-t-il ?

Exercices

1. Le code qui suit n'est pas rédigé correctement. Saisissez-le et compilez-le pour voir le résultat.

```
#include <stdio.h>
#include <stdlib.h>
int x,y;int main() { printf(
"\nEntrez deux nombres");scanf(
"%d %d",&x,&y);printf(
"\n\n%d est plus grand",(x>y)?x:y); exit(EXIT_SUCCESS);}
```

2. Reprenez le code de l'exercice 1 pour le rendre plus clair.
3. Transformez le Listing 4.1 pour compter en montant plutôt qu'en descendant.
4. Écrivez une instruction `if` qui donne la valeur de `x` à `y` si `x` se situe entre 1 et 20. Dans le cas contraire, ne pas changer la valeur de `y`.
5. Utilisez l'opérateur de condition pour faire l'exercice précédent.
6. Transformez les instructions suivantes pour n'obtenir qu'une instruction `if` avec des opérateurs composés.

```
if (x < 1)
    if (x > 10)
          instruction;
```

7. Quelles sont les valeurs des expressions suivantes :
   a) `(1 + 2 * 3)`.
   b) `10 % 3 * 3   (1 + 2)`.
   c) `((1 + 2) * 3)`.
   d) `(5 == 5)`.
   e) `(x = 5)`.
8. Si `x = 4`, `y = 6`, et `z = 2`, le résultat aux questions suivantes est-il vrai ou faux :
   a) `if (x == 4)`.
   b) `if (x != y   z)`.
   c) `if (z = 1)`.
   d) `if (y)`.
9. Écrivez une instruction `if` pour savoir si une personne est un adulte (21 ans), mais pas une personne âgée (65 ans).
10. **CHERCHEZ L'ERREUR** : Corrigez le programme suivant.

```
/* programme bogué */
#include <stdio.h>
#include <stdlib.h>
int x = 1:
int main()
{
    if (x = 1);
        printf("x égal 1");
    sinon
        printf("x n'est pas égal à 1");
   exit(EXIT_SUCCESS)
}
```

Exemple pratique 2

Le nombre mystère

Voici la seconde section de ce type. Vous savez que son objectif est de présenter un programme complet plus fonctionnel que les exemples des chapitres. Ce programme comporte quelques éléments non encore étudiés. Mais il ne devrait pas être trop difficile à comprendre. Prenez le temps de tester le code en le modifiant éventuellement et en observant les résultats. Attention aux fautes de frappe qui ne manqueront pas de provoquer des erreurs de compilation.

Listing Exemple pratique 2 : trouver_nombre.c

```
 /* Programme : trouver_nombre.c
  * Objectif : Ce programme choisit un nombre de façon aléatoire
  *            et demande à l'utilisateur de le retrouver
  * Retour :   aucun
  */

 #include <stdio.h>
 #include <stdlib.h>
 #include <time.h>

#define NO   0
#define YES  (!NO)

int main( void )
{
    int guess_value = -1;
    int number;
    int nbr_of_guesses;
    int done = NO;

    printf("Sélection d'un nombre aléatoire\n");
```

Listing Exemple pratique 2 : trouver_nombre.c ***(suite)***

```

    /* le temps entre dans le calcul du nombre aléatoire */
    srand( time( NULL ) );
    number = rand();

    nbr_of_guesses = 0;
    while ( done == NO )
    {
         printf("\nDonnez un nombre entre 0 et %d> ", RAND_MAX);
         scanf( "%d", &guess_value );  /* lecture du nombre */

         nbr_of_guesses++;

         if ( number == guess_value )
         {
             done = YES;
         }
         else
         if ( number < guess_value )
         {
             printf("\nCe nombre est trop grand !");
         }
         else
         {
             printf("\nCe nombre est trop petit !");
         }
    }

    printf("\n\nFélicitations! Vous avez trouvé en %d essais !",
             nbr_of_guesses);
    printf("\n\nLa réponse était %d\n\n", number);
    exit(EXIT_SUCCESS);
}
```

Analyse

Ce programme vous demande simplement de deviner le nombre choisi de façon aléatoire par l'ordinateur. À chaque essai, il vous indique si votre chiffre est supérieur ou inférieur à la solution. Lorsque vous trouvez le résultat, le programme vous indique le nombre de tentatives qui ont été nécessaires pour y arriver.

Vous pouvez tricher en ajoutant une ligne qui affichera le nombre sélectionné :

```
printf("le nombre choisi (réponse) est : %d", number);
```

Vous pourrez aussi contrôler avec cette ligne le bon fonctionnement du programme. N'oubliez pas de la supprimez si vous le transmettez à des amateurs de jeux.

5

Les fonctions

Les fonctions sont au centre de la programmation du langage C et de la philosophie du développement dans ce langage. Nous avons vu les fonctions de bibliothèque qui sont fournies avec le compilateur. Ce chapitre traite des fonctions utilisateur qui sont créées par le programmeur.

Aujourd'hui, vous allez apprendre :

- De quoi est constituée une fonction
- Les avantages de la programmation structurée avec ces fonctions
- Comment créer une fonction
- Les déclarations de variables locales d'une fonction
- Comment transmettre une valeur de la fonction vers le programme
- Comment passer des arguments à une fonction

Qu'est-ce qu'une fonction ?

Ce chapitre aborde les fonctions sous deux angles, en les définissant et en montrant à quoi elles servent.

Définition

Une *fonction* est un bloc de code C indépendant, référencé par un nom, qui réalise une tâche précise et qui peut renvoyer une valeur au programme qui l'a appelée. Examinons cette définition :

- *Une fonction est référencée par un nom*. Ce nom est unique et en l'introduisant dans le source de votre programme, vous pouvez exécuter le code de la fonction. Une fonction peut être appelée par une autre fonction.
- *Une fonction est indépendante*. Une fonction peut effectuer sa tâche avec ou sans échanges avec une autre partie du programme.
- *Une fonction réalise une tâche particulière*. La tâche est l'unité de base du travail réalisé par le programme. Cela peut être l'envoi d'une ligne de texte vers l'imprimante, un tri, ou le calcul d'une racine carrée.
- *Une fonction peut renvoyer une valeur au programme appelant*. Quand ce programme appelle la fonction, le code de cette fonction est exécuté. Ces instructions peuvent renvoyer une information au programme.

Exemple de fonction

Listing 5.1 : Ce programme emploie une fonction utilisateur pour calculer le cube d'un nombre

```
/* Exemple d'une fonction simple */
#include <stdio.h>
#include <stdlib.h>
long cube(long x);

long input, reponse;

int main()
{
   printf("Entrez une valeur entière : ");
   scanf("%d", &input);
   reponse = cube(input);
```

```
13:    /* Note: %ld est la spécification de conversion d'un entier long */
14:    printf("\n\nLe cube de %ld est %ld\n.", input, reponse);
15:    exit(EXIT_SUCCESS);
16: }
17:
18: long cube(long x)
19: {
20:    long x_cube;
21:
22:    x_cube = x * x * x;
23:    return x_cube;
24: }
```



```
Entrez un nombre entier : 100
Le cube de 100 est 1000000
Entrez un nombre entier : 9
Le cube de 9 est 729
Entrez un nombre entier : 3
Le cube de 3 est 27
```



L'analyse suivante ne porte pas sur la totalité du programme, mais se concentre sur les éléments du programme directement en relation avec la fonction.

Analyse

La ligne 4 contient le *prototype* (déclaration) de la fonction qui représente un modèle pour une fonction qui apparaîtra plus loin dans le programme. Ce prototype contient le nom de la fonction, la liste des variables qui lui seront transmises, et éventuellement le type de variable que la fonction renverra. La ligne 4 nous indique que la fonction s'appelle `cube`, qu'elle utilise une variable de type `long`, et qu'elle renverra une variable de type `long`. Les variables qui sont transmises à la fonction sont des arguments et apparaissent entre parenthèses derrière le nom de la fonction. Dans notre exemple, l'argument de la fonction est `long x`. Le mot clé situé avant le nom indique le type de variable renvoyé par la fonction. Dans notre cas, c'est un type long.

La ligne 12 appelle la fonction `cube` et lui transmet en argument la variable `input`. La valeur renvoyée par la fonction est stockée dans la variable `reponse`. Ces deux variables sont déclarées en ligne 6 avec le type `long` ce qui correspond bien au prototype de la ligne 4.

Les lignes 18 à 24, qui constituent la fonction `cube` elle-même, sont appelées *définition de la fonction*. La fonction commence par un *en-tête* en ligne 18 qui indique son nom (dans notre exemple `cube`). Cet en-tête décrit aussi le type de donnée qui sera renvoyée et les

arguments. Vous pouvez remarquer que l'en-tête de fonction est identique au prototype (le point-virgule en moins).

Les lignes 20 à 23 représentent le corps de la fonction ; il est encadré par des accolades. Il contient des instructions, comme celle de la ligne 22, qui sont exécutées chaque fois que la fonction est appelée. La ligne 20 est une déclaration de variable comme nous en avons déjà vu, à une exception près : c'est une variable locale. Les variables locales, qui sont traitées au Chapitre 12, sont déclarées à l'intérieur d'une fonction. La ligne 23 indique la fin de la fonction et renvoie une valeur au programme appelant. Dans notre cas, c'est la valeur de `x cube` qui est transmise.

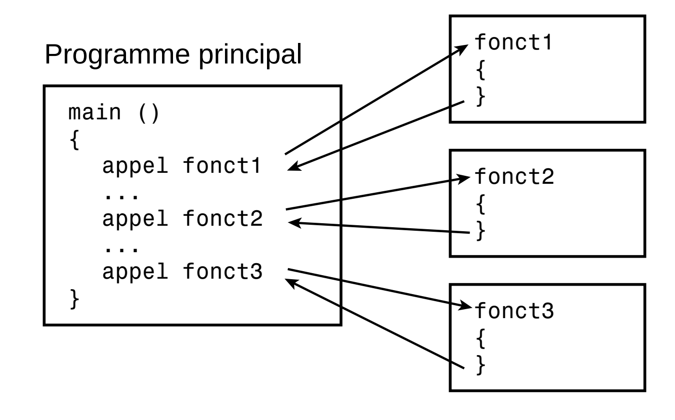
Si vous comparez la structure de la fonction avec celle de la fonction `main()`, vous ne trouverez pas de différence. Comme `printf()` ou `scanf()` qui sont des fonctions de bibliothèque, toutes les fonctions peuvent recevoir des arguments et renvoyer une valeur au programme qui les a appelées.

Fonctionnement

Les instructions d'une fonction dans un programme ne sont exécutées que lorsqu'elles sont appelées par une autre partie de ce programme. Quand la fonction est appelée, le programme lui transmet des informations sous la forme d'arguments. Un argument est une donnée de programme dont la fonction a besoin pour exécuter sa tâche. La fonction s'exécute puis le programme reprend à partir de la ligne qui contient l'appel en récupérant éventuellement une valeur de retour.

La Figure 5.1 représente un programme qui appelle trois fonctions. Les fonctions peuvent être appelées autant de fois que nécessaire et dans n'importe quel ordre.

Figure 5.1
Quand un programme appelle une fonction, les instructions de la fonction s'exécutent puis le programme reprend la main pour la suite de son exécution.



Fonctions

Prototype de la fonction

```
type_retour nom_fonction (type_arg nom-1, ..., type_arg nom-n);
```

Définition de la fonction

```
type_retour nom_fonction (type_arg nom-1, ..., type_arg nom-n);
{
     /* instructions; */
}
```

Le *prototype de la fonction* fournit au compilateur la description d'une fonction qui est définie plus loin dans le programme. Cette description comprend le nom de la fonction, le type de valeur (`type retour`) qui sera renvoyée à la fin de l'exécution de la fonction et les types d'arguments (`type arg`) qui lui seront transmis. Il peut éventuellement contenir le nom des variables qui seront transmises et il se termine toujours par un point-virgule.

La *définition de la fonction* est la fonction, c'est-à-dire le code qui sera exécuté. La première ligne, *l 'en-tête de la fonction,* est identique au prototype, à l'exception du point-virgule. Cet en-tête doit obligatoirement contenir le nom des variables arguments qui était en option dans le prototype. Il est suivi du corps de la fonction, les instructions entre accolades. Si le `type retour` n'est pas `void`, il faut une instruction `return` qui renvoie une valeur correspondant au `type retour`.

Exemples de prototypes

```
double carré (double nombre);
void impression_rapport (int nombre_rapport);
int choix_menu (void);
```

Exemples de définitions

```
double carré (double nombre)    /* en-tête de fonction */
{                               /* accolade de début */
     return (nombre * nombre);  /* corps de la fonction */
}                               /* accolade de fin */
impression_rapport (int nombre_rapport)
{
     if (nombre_rapport == 1)
         puts ("Impression rapport 1");
      else
         puts("pas d'impression du rapport 1");
}
```

Les fonctions et la programmation structurée

En utilisant des fonctions dans votre programme vous pratiquez la *programmation structurée*. Cela signifie que les différentes tâches du programme sont réalisées par des portions de code indépendantes : les fonctions.

Avantages de la programmation structurée

La programmation structurée a deux avantages :

- Il est plus facile d'écrire un programme structuré, car des problèmes de programmation complexes peuvent être divisés en tâches plus simples. Chacune de ces tâches est réalisée par une fonction dont le code et les variables sont indépendants du reste du programme.
- Un programme structuré est plus facile à corriger. En effet, l'erreur pourra facilement être localisée dans une partie spécifique du code (celle de la fonction qui ne s'exécute pas correctement).

Ce type de programmation peut vous faire gagner du temps. En effet, une fonction que vous aurez écrite dans un programme pour réaliser une certaine tâche, pourra facilement être utilisée dans un autre programme pour réaliser la même tâche. Si celle-ci est légèrement différente, il sera plus facile de la modifier que d'en créer une nouvelle.

Étude d'un programme structuré

Pour écrire un programme structuré, vous devez faire une étude préalable avant de créer la première ligne de code. Cette étude doit être composée de la liste des tâches à réaliser par le programme. Par exemple, pour écrire un programme de gestion de vos adresses, voici les tâches que devrait pouvoir faire ce programme :

- Saisir les nouveaux noms et adresses.
- Modifier un enregistrement.
- Trier les noms de famille.
- Imprimer les noms et adresses sur des enveloppes.

Cette liste permet de partager le programme en quatre fonctions principales. Vous pouvez maintenant décomposer ces tâches en sous-tâches plus simples. "Saisir les nouveaux noms et adresses" peut ainsi devenir :

- Lire la liste des adresses existantes.

- Demander à l'utilisateur de taper une ou plusieurs nouvelles adresses.
- Ajouter les nouveaux enregistrements.
- Sauvegarder la nouvelle liste sur disque.

De la même façon, vous pouvez décomposer "Modifier un enregistrement" :

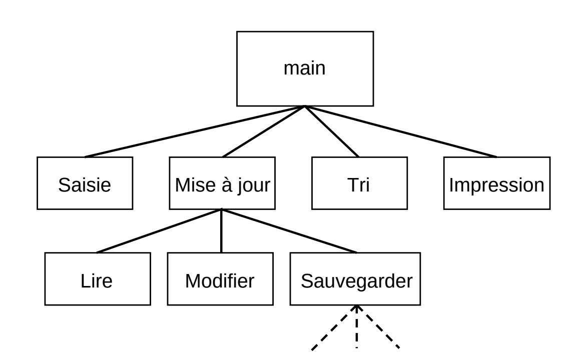
- Lire la liste des adresses existantes.
- Modifier un ou plusieurs enregistrements.
- Sauvegarder la nouvelle liste sur disque.

Ces deux listes ont deux sous-tâches en commun : celle qui lit et celle qui sauvegarde. Vous pouvez écrire une fonction pour "lire les adresses existantes sur disque" et cette fonction sera appelée par chacune des fonctions "Saisir les nouveaux noms et adresses" et "Modifier un enregistrement". Le raisonnement est le même pour "Sauvegarder la nouvelle liste sur disque".

En identifiant ainsi des parties du programme qui réalisent la même tâche, vous pouvez écrire des fonctions qui seront appelées plusieurs fois par le programme. Vous gagnez du temps et votre programme est plus efficace.

Comme le montre la figure suivante, cette méthode de programmation conduit à une structure de programme hiérarchique.

Figure 5.2
Un programme structuré est organisé de façon hiérarchique.



L'approche *top-down*

Avec la programmation structurée, les programmeurs adoptent une approche *top-down* (de haut en bas). Cela est illustré par la Figure 5.2 ou la structure du programme ressemble à

un arbre inversé. En général, les tâches importantes sont réalisées au bout des "branches", les autres fonctions sont là pour orienter l'exécution du programme vers ces fonctions.

Par ce fait, beaucoup de programmes C ont très peu de code dans la partie principale `main()`, et une partie de code importante pour les fonctions. Vous trouverez dans la partie `main` quelques lignes d'instructions dont le rôle se limitera à aiguiller l'exécution du programme vers les fonctions. Ces lignes représentent souvent un menu dans lequel l'utilisateur choisit la tâche qu'il veut réaliser. Chaque partie de ce menu utilise une fonction différente.

Le Chapitre 13, avec l'instruction `switch`, vous apprendra à créer un bon système à base de menus.



À faire

Établir le plan du programme avant de commencer à coder. En déterminant à l'avance la structure de votre programme, vous gagnerez du temps au développement et à la correction.

À ne pas faire

Essayer de tout faire avec une seule fonction. Une fonction doit exécuter une seule tâche, comme lire un fichier ou sauvegarder des données.

Écriture d'une fonction

La première étape de l'écriture d'une fonction consiste à savoir ce que doit faire cette fonction. La suite ne représente pas de difficultés particulières.

En-tête

La première ligne d'une fonction est l'en-tête de la fonction. Il est constitué de trois éléments qui ont chacun un rôle particulier.

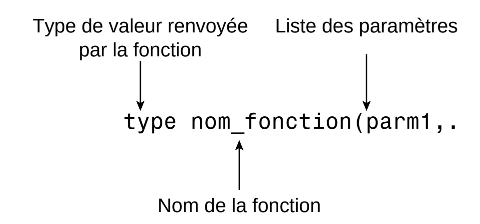


Figure 5.3
Les trois composants de l'en-tête de fonction.

Type de la valeur renvoyée

Le type de la valeur renvoyée indique le type de donnée que la fonction retournera au programme appelant. Ce type peut être : `char`, `int`, `long`, `float`, ou `double`. Vous pouvez aussi créer une fonction qui ne retourne pas de valeur en utilisant le type de valeur renvoyée `void`. Voici quelques exemples :

```
int fonc1(...)    /* renvoie une donnée de type int. */
float fonc2(...)  /* renvoie une donnée de type float. */
void fonc3(...)   /* ne renvoie aucune donnée. */
```

Nom

Le nom d'une fonction doit suivre les mêmes règles que les noms de variables C (voir Chapitre 3) et il doit être unique (vous ne pouvez pas appeler une variable ou une autre fonction par le même nom).

Liste des paramètres

Beaucoup de fonctions utilisent des arguments, et elles ont besoin de connaître le type de donnée qu'elles vont recevoir. Vous pouvez transmettre à une fonction n'importe quel type de donnée C en l'indiquant dans l'en-tête de cette fonction avec la liste de paramètres.

À chaque argument transmis vers la fonction doit correspondre une entrée dans la liste de paramètres. Voici, par exemple, l'en-tête de la fonction du Listing 5.1 :

```
long cube (long x)
```

La liste de paramètres contient `long x` ce qui indique à la fonction qu'elle recevra un paramètre de type `long` représenté par la variable `x`. Si la liste contient plusieurs paramètres, ils sont séparés par une virgule. Examinons l'en-tête suivant :

```
void fonction1 (int x, float y, char z)
```

La fonction va recevoir trois arguments : `x` de type `int`, `y` de type `float`, et `z` de type `char`. Une fonction qui ne doit pas recevoir de paramètre a une liste d'arguments qui contient `void` :

```
void fonction2 (void)
```



Il ne faut pas mettre de point-virgule à la fin de votre en-tête de fonction, sinon le compilateur générera un message d'erreur.

Il ne faut pas confondre paramètre et argument. Un paramètre est une entrée de l'en-tête de la fonction, il "annonce" l'argument. Les paramètres ne peuvent changer pendant l'exécution du programme.

Un argument est une donnée transmise à la fonction par le programme appelant. Chaque fois que la fonction sera appelée, le programme pourra lui transmettre des valeurs d'arguments différents. Par contre, le nombre et le type des arguments transmis ne doivent pas changer. L'argument est référencé par le nom correspondant dans la liste de paramètres.

Listing 5.2 : Différence entre argument et paramètre

```
/*   Ce programme montre la différence entre paramètre et argument   */
#include <stdio.h>
#include <stdlib.h>

float x = 3.5, y = 65.11, z;

float moitie_de(float k);

int main()
{
/* Dans cet appel, x est l'argument de moitie_de(). */
    z = moitie_de(x);
    printf("La valeur de z = %f\n", z);

/* Dans cet appel,y est l'argument de moitie_de(). */
    z = moitie_de(y);
    printf("La valeur de z = %f\n", z);
    exit(EXIT_SUCCESS);
}

float moitie_de(float k)
{
   /* k est le paramètre. Chaque fois que moitie_de est */
   /* appelée, k prend la valeur qui a été passée en argument */

    return (k/2);
}
```



```
La valeur de z = 1.750000
La valeur de z = 32.555000
```

Figure 5.4
Schéma des relations entre arguments et paramètres.

Premier appel de la fonction `z=moitié_de(x);` 3,5 `float moitié_de(float k)`

Second appel de la fonction `z=moitié_de(y);` 65,11 `float moitié_de(float k)`

Analyse

Examinons le programme du Listing 5.2. Le prototype de la fonction `moitie de()` se trouve en ligne 7. Les lignes 12 et 16 appellent cette fonction dont les instructions sont en lignes 21 à 27. La ligne 12 envoie l'argument `x`, qui contient la valeur 3,5 ; la ligne 16 envoie l'argument `y`, qui contient la valeur 65,11. Les résultats donnés par le programme donnent bien la moitié de ces deux valeurs. Les valeurs contenues dans `x` et `y` sont transmises par l'intermédiaire de l'argument `k` de `moitie de()`. On a fait, en fait, une copie de `x` en `k` puis une copie de `y` en `k`. La fonction a ensuite retourné ces valeurs divisées par deux.



À faire

Choisir des noms de fonctions qui indiquent ce qu'elles font.

À ne pas faire

Transmettre à une fonction une donnée dont elle n'a pas besoin.

Transmettre plus (ou moins) d'arguments qu'il n'y a de paramètres. En langage C, le nombre d'arguments transmis doit correspondre exactement au nombre de paramètres.

Instructions

Le *corps de la fonction* se trouve entre accolades à la suite de l'en-tête. Quand la fonction est appelée, l'exécution commence à la première instruction du corps de la fonction et se termine à la première instruction `return` rencontrée ou à l'accolade de fin.

Les variables locales

Il est possible de déclarer des variables internes à la fonction, ce sont les *variables locales*. Elles sont particulières à cette fonction et différentes de variables du même nom qui pourraient se trouver dans une autre partie du programme.

La déclaration d'une variable locale suit les mêmes principes que ceux d'une variable standard qui ont été énoncés dans le Chapitre 3. Vous pouvez les initialiser au moment de la déclaration et elles peuvent être n'importe quel type de variable C. Voici un exemple de quatre variables déclarées dans une fonction :

```
int fonction1(int y)
{
    int a, b = 10;
    float taux;
    double cout = 12.55;
    /* instructions… */.
}
```

Cette déclaration crée les variables locales `a`, `b`, `taux`, et `cout` qui seront utilisées par le code de la fonction. Les paramètres de la fonction sont considérés comme des déclarations de variable. Si la liste n'est pas vide, ces variables sont donc disponibles pour les instructions qui suivent.

Le Listing 5.3 vous donne la démonstration de l'indépendance des variables locales vis à vis des autres variables du programme.

Listing 5.3 : Utilisation de variables locales

```
 1: /* Démonstration de l'indépendance des variables locales. */
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4:
 5: int x = 1, y = 2;
 6:
 7: void demo(void);
 8:
 9: int main()
10: {
11:    printf("Avant d'appeler demo(), x = %d et y = %d.\n", x, y);
12:    demo();
13:    printf("Après l'appel de demo(), x = %d et y = %d\n.", x, y);
14:    exit(EXIT_SUCCESS);
15: }
16:
17: void demo(void)
18: {
19:    /* Déclaration et initialisation de deux variables locales. */
20:
21:    int x = 88, y = 99;
22:
23:    /* Affichage de leur valeur. */
24:
25:    printf("Dans la fonction demo(), x = %d et y = %d.\n", x, y);
26: }
```

```
Avant d'appeler demo(), x = 1 et y = 2.
Dans la fonction demo(), x = 88 et y = 99.
Après l'appel de demo(), x = 1 et y = 2.
```

Analyse

La ligne 5 de ce programme déclare les variables `x` et `y`. Elles sont déclarées en dehors de toute fonction, ce sont donc des variables globales. La ligne 7 contient le prototype de la fonction `demo()`. Cette ligne commence par `void` qui indique que la fonction ne travaille pas avec des paramètres. Le deuxième `void` qui remplace le type de paramètre indique que la fonction ne retourne aucune valeur au programme appelant. La fonction principale

`main()` commence en ligne 9 : la fonction `printf()` de la ligne 11 affiche les valeurs de `x` et `y` puis la fonction `demo()` est appelée. `demo()` déclare ses versions locales de `x` et `y` en ligne 21. La ligne 24 démontre que les valeurs des variables locales supplantent celles des variables du programme. Après l'appel de la fonction, la ligne 13 affiche de nouveau `x` et `y` qui sont revenues à leur valeur initiale.

Nous venons de vous démontrer que les variables locales `x` et `y` de la fonction étaient totalement indépendantes des variables globales `x` et `y` des autres parties du programme. L'utilisation des variables dans une fonction doit suivre trois règles :

- Pour utiliser une variable dans une fonction, vous devez la déclarer dans l'en-tête ou le corps de la fonction.
- Pour que la fonction puisse recevoir une donnée du programme appelant, celle-ci doit être transmise en argument.
- Pour que le programme appelant puisse recevoir une valeur retour de la fonction, cette valeur doit être explicitement renvoyée par la fonction.

Créer des variables locales est une des façons de rendre les fonctions indépendantes. Une fonction peut effectuer toutes sortes de manipulations de données avec des variables locales et cela ne pourra pas affecter une autre partie du programme.

Les instructions

Il n'y a aucune contrainte pour l'utilisation des instructions dans une fonction. La seule chose que l'on ne puisse pas faire dans une fonction est de définir une autre fonction. Vous pouvez utiliser toutes les autres instructions du langage C en incluant les boucles (traitées au Chapitre 6), les instructions `if`, ou les instructions d'affectation.

Il n'existe pas non plus de contrainte de taille , mais en programmation structurée une fonction effectue une tâche simple. Si la fonction que vous créez est longue, vous essayez peut-être d'exécuter une tâche trop complexe pour une seule fonction. Celle-ci pourrait être décomposée en plusieurs fonctions plus petites.

En général, la taille maximum d'une fonction est de 25 à 30 lignes de code. Certaines seront plus longues, d'autres n'auront besoin que de quelques lignes. Lorsque vous aurez une bonne expérience de programmation, vous saurez si une fonction doit être divisée en plusieurs plus petites ou non.

Comment renvoyer une valeur

Le mot clé `return` permet de renvoyer une valeur au programme appelant. Quand une instruction `return` est rencontrée au cours de l'exécution de la fonction, celle-ci est évaluée et la valeur trouvée est transmise au programme.

Examinons l'exemple suivant :

```
int fonct1(int var)
{
    int x;
    /* instructions... */
    return x;
}
```

Quand cette fonction est appelée, toutes les instructions s'exécutent jusqu'à l'instruction `return` qui renvoi la valeur `x` au programme appelant. L'expression qui suit le mot clé `return` peut être n'importe quelle expression du langage C.

Une fonction peut contenir plusieurs instructions `return`. C'est la première qui est exécutée qui pourra provoquer le retour vers le programme.

Listing 5.4 : Utilisation de plusieurs instructions return dans une fonction

```
/* Exemple de plusieurs instructions return dans une fonction. */
#include <stdio.h>
#include <stdlib.h>

int x, y, z;

int larger_of(int a, int b);

int main()
{
   puts("Entrez deux valeurs entières différentes : ");
   scanf("%d%d", &x, &y);

   z = larger_of(x,y);

   printf("\nLa valeur la plus grande est %d.", z);
   exit(EXIT_SUCCESS);
}

int larger_of(int a, int b)
{
   if (a > b)
       return a;
   else
       return b;
}
```

```
Entrez 2 valeurs entières différentes :
200 300

La valeur la plus grande est 300.

Entrez 2 valeurs entières différentes :
300
200
La valeur la plus grande est 300.
```

Analyse

Le fichier en-tête stdio.h est appelé ligne 3 pour permettre les entrées/sorties demandées par le programme. La ligne 7 est le prototype de la fonction `larger of()`. Vous pouvez remarquer qu'elle reçoit deux variables `int` comme paramètres et qu'elle renverra une valeur entière. Cette fonction est appelée en ligne 14 avec `x` et `y`. Elle compare `a` et `b` en ligne 22. Si `a` est plus grand que `b`, la ligne 23 exécute l'instruction `return` et la fonction se termine : les lignes 24 et 25 seront ignorées. Si `b` est plus grand que `a`, l'exécution de la fonction se poursuit à la ligne 24 avec la clause `else` et la ligne 25 exécute la seconde instruction `return`. Cette fonction possède bien plusieurs instructions `return` qui sont exécutées en fonction de la valeur des variables transmises lors de l'appel.

La dernière remarque à faire sur ce programme concerne la ligne 11. Elle contient la nouvelle fonction `puts()` (lire *put string*, en français *envoi chaîne de caractères*). Cette fonction, qui est traitée au Chapitre 10, envoi simplement une chaîne de caractères vers la sortie standard (en général l'écran).

Retenez que la valeur de la variable renvoyée par une fonction doit correspondre au type de variable déclaré dans l'en-tête de la fonction et dans le prototype.



En programmation structurée, vous devez avoir seulement une entrée et une sortie dans une fonction. Cela signifie que vous devez vous efforcer d'obtenir une seule instruction `return`. Un programme sera cependant plus facile à interpréter avec plusieurs de ces instructions. Dans ce cas, la priorité sera donnée à l'interprétation.

Prototype

Un programme peut contenir un prototype pour chaque fonction qu'il utilise. La ligne 4 du Listing 5.1 vous donne un exemple de prototype. Qu'est-ce qu'un prototype et à quoi sert-il ?

Nous avons vu que le prototype est identique à l'en-tête de la fonction et qu'il se termine par un point-virgule. Il contient des informations sur le nom, les paramètres et le type de donnée renvoyée par la fonction. Ces informations sont destinées au compilateur. Il pourra ainsi vérifier, à chaque appel de la fonction, que vous avez transmis le bon nombre et le bon type de paramètres, et que la valeur de retour uti-lisée est correcte. Si ce contrôle échoue, le compilateur envoi un message d'erreur. Le prototype est obligatoire si la fonction est définie après un appel à celle-ci. Sinon, le compilateur n'en a pas connaissance et signale une erreur.

Un prototype de fonction peut ne pas être strictement identique à l'en-tête de la même fonction. Le nom des paramètres peut être différent du moment que le nombre, le type, et l'ordre

est respecté. Le plus simple pour le programmeur reste bien sûr d'utiliser le copier-coller de l'éditeur de texte pour copier l'en-tête de la fonction et créer le prototype en ajoutant le point-virgule. Le risque d'erreur sera réduit et votre programme gagnera en clarté.

Pour des raisons pratiques, il est préférable de grouper tous les prototypes au même endroit, avant le début de la fonction principale `main()` voire dans un fichier d'en-têtes (extension .h) à inclure avec #include.



À faire

Utiliser des variables locales quand c'est possible.

Créer des fonctions qui exécutent une seule tâche.

À ne pas faire

Renvoyer une valeur dont le type ne correspond pas à celui de la fonction.

Écrire des fonctions trop longues : essayez de les découper en plusieurs tâches plus petites.

Créer des fonctions avec plusieurs instructions `return` alors que ce n'est pas nécessaire.

Passage d'arguments à une fonction

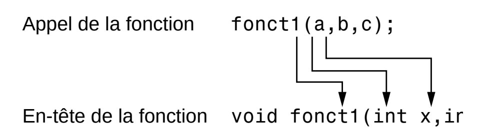
Les arguments sont transmis à une fonction au moyen de la liste entre parenthèses qui suit le nom de la fonction. Le nombre et le type de ces arguments doivent correspondre aux indications de rendu de l'en-tête et du prototype de la fonction. Si vous essayez de transmettre un mauvais nombre ou un type incorrect d'arguments, le compilateur le détectera à partir des informations fournies par le prototype.

Dans le cas où une fonction reçoit plusieurs valeurs, les arguments qui sont listés dans l'appel de la fonction sont attribués dans le même ordre aux paramètres de cette fonction comme l'indique la Figure 5.5.

Un argument peut être n'importe quelle expression du langage C : une constante, une variable, une expression mathématique ou logique, ou une autre fonction (avec une valeur de retour). Par exemple, si `moitie()`, `carre()`, et `troisieme()` sont des fonctions qui renvoient une valeur vous pouvez écrire :

```
x = moitie(troisieme(carre(moitie(y))));
```

Figure 5.5
Les arguments sont attribués dans l'ordre aux paramètres.



La première fonction appelée par le programme sera `moitie()` à qui ont passera l'argument `y`. La valeur de retour sera transmise à la fonction `carre()` puis la fonction `troi sieme()` sera appelée en lui transmettant la valeur renvoyée par `carre()`. Enfin, la fonction `moitie()` sera de nouveau appelée avec la valeur de retour de `troisieme()` en argument. C'est la valeur renvoyée par `moitie()` qui sera finalement attribuée à `x`. Le code suivant est équivalent à l'exemple précédent :

```
a = moitie(y);
b = carre(a);
c = troisieme(b);
x = moitie(c);
```

Appel d'une fonction

Il existe deux façons d'appeler une fonction. La première consiste à utiliser son nom suivi de la liste d'arguments entre parenthèses, dans une instruction. Si la fonction a une valeur de retour, elle sera ignorée.

```
wait(12);
```

La seconde méthode ne concerne que les fonctions qui renvoient une valeur. Ces fonctions étant évaluées à leur valeur de retour, elles font partie des expressions du langage C et peuvent donc être utilisées comme tel :

```
printf("La moitié de %d est %d.", x, moitie_de(x));
```

L'exécution de cette instruction commence par l'appel de la fonction `moitie de()` avec la valeur de `x`. La fonction `printf()` est ensuite exécutée avec les valeurs `x` et `moitie de(x)`.

Voici un autre exemple de plusieurs fonctions utilisées comme des expressions :

```
y = moitie_de(x) +  moitie_de(z);
```

qui est l'équivalent de :

```
a = moitie_de(x);
b = moitie_de(z);
y = a + b;
```

Ces deux derniers exemples vous montrent comment utiliser les valeurs renvoyées par les fonctions.

```
if (moitie_de(x) > 10)
{
    /* instructions */
}
```

Si la valeur calculée par la fonction remplit la condition, l’instruction `if` est vrai et les instructions sont exécutées. Dans le cas contraire, les instructions ne sont pas exécutées.

```
if (processus() != OK)
{
    /* instructions */    /* traitement des erreurs */
}
```

Dans cet exemple, la valeur renvoyée par le processus est contrôlée pour savoir si l’exécution de ce processus s’est déroulée correctement. Dans le cas contraire, les instructions traiteront les erreurs. Cette méthode est souvent employée pour lire des fichiers, comparer des valeurs et allouer de la mémoire.

Le compilateur générera un message d’erreur si vous tentez d’utiliser une fonction avec un type retour `void` comme une instruction.



À faire

Transmettre des paramètres aux fonctions pour rendre leur code facilement réutilisable.

Tirer parti de la possibilité de remplacer une expression par une fonction.

À ne pas faire

Rendre une instruction illisible en y introduisant trop de fonctions.

Récurrence

Lorsqu’une fonction s’appelle elle-même de façon directe ou indirecte, on parle de *récurrence*. Dans une récurrence indirecte, une fonction appelle une autre fonction qui appelle à son tour la première. Le langage C permet ce schéma qui peut se révéler très utile dans certaines circonstances.

Par exemple, la récurrence peut être utilisée pour calculer le factoriel d’un nombre. Le factoriel d’un nombre `x` se note `x!` et se calcule de la façon suivante :

```
x! = x * (x-1) * (x-2) * (x-3) *, etc. * (2) * 1
```

Voici une autre méthode pour calculer `x!` :

```
x! = x * (x-1)!
```

ou

```
(x-1)! = (x-1) * (x-2)!
```

Vous pouvez continuer à calculer de façon récurrente jusqu'à la valeur 1. Le programme du Listing 5.5 utilise une fonction récurrente pour calculer les factorielles. Le programme n'utilisant que les entiers non signés, la valeur transmise au programme sera limitée à 8. Le factoriel de 9 ou de nombres plus grands sort des limites autorisées pour les entiers.

Listing 5.5 : Calcul des factorielles avec une fonction récurrente

```
/* Exemple de fonction récurrente. Calcul de la factorielle d'un nombre
#include <stdio.h>
#include <stdlib.h>

unsigned int f, x;
unsigned int factorielle(unsigned int a);

int main()
{
   puts("Entrez une valeur entière entre 1 et 8: ");
   scanf("%d", &x);

   if(x > 8 || x < 1)
   {
      printf("On a dit entre 1 to 8 !");
   }
   else
   {
      f = factoriel(x);
      printf("Factorielle %u égal %u\n", x, f);
   }
   exit(EXIT_SUCCESS);
}

unsigned int factorielle(unsigned int a)
{
   if (a == 1)
       return 1;
    else
   {
       a *= factorielle(a-1);
       return a;
    }
}
```



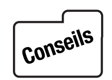
```
Entrez une valeur entière entre 1 et 8 :
6
Factorielle 6 égal 720
```

Analyse

La première partie du programme ressemble à celles des programmes que nous avons déjà étudiés. Les commentaires sont en ligne 1, l'appel du fichier en-tête en ligne 3 et la ligne 5 contient la déclaration de deux valeurs entières non signées. Le prototype de la fonction `factorielle()` se trouve en ligne 6. La fonction principale `main()` est constituée des lignes 8 à 23. Les lignes 10 et 11 permettent de récupérer le nombre choisit par l'utilisateur.

L'instruction `if` des lignes 13 à 21 est intéressante. Elle permet de contrôler la valeur entrée par l'utilisateur. Si elle est plus grande que 8, un message d'erreur est envoyé. Sinon, le calcul de la factorielle se fait en ligne 19 et la ligne 20 affiche le résultat. Traitez les erreurs de cette façon chaque fois que vous suspectez un problème (comme la taille d'un nombre par exemple).

Notre fonction récurrente se trouve lignes 25 à 34. La valeur qui lui est transmise est attribuée à `a` et contrôlée en ligne 27. Si la valeur de `a` est 1, le programme renvoi la valeur 1. Si la valeur de `a` est différente de 1, on attribue à `a` la valeur de `a * factorielle(a 1)`. Le programme appelle la fonction factoriel de nouveau, mais cette fois la valeur de `a` est `(a 1)`. Si `(a 1)` est différent de 1, `factorielle()` est appelé de nouveau avec `((a 1) 1)`. Le processus se poursuit jusqu'à ce que l'ordre `if` de la ligne 27 soit vrai.



À faire

Vous exercer à la récurrence avant de l'utiliser.

À ne pas faire

Ne pas utiliser la récurrence s'il y a plusieurs itérations (répétition d'une instruction). En effet, la récurrence mobilise beaucoup de ressources pour que la fonction puisse se gérer.

Le placement des fonctions

Les définitions de fonctions peuvent se placer dans le même fichier source que la fonction principale `main()` et après la dernière instruction de celle-ci.

Vous pouvez sauvegarder vos fonctions utilisateur dans un fichier séparé de celui qui contient la fonction `main()`. Cette technique est très utile dans le cas d'un programme très

long ou si vous voulez utiliser cette même série de fonctions dans un autre programme (voir Chapitre 21).

Figure 5.6
Structure d'un programme qui utilise des fonctions.

```
/*  Début de code source */
    ...
    prototype des fonctions
    ...
    int main()
    {
       ...
       ...
    }
    fonct1()
    {
       ...
    }
    fonct2()
    {
       ...
    }
/*  Fin du code source */
```

Résumé

Les fonctions sont une partie importante de la programmation en langage C. Le code qui les représente est indépendant du programme. Le rôle d'une fonction est d'exécuter une tâche particulière : quand votre programme a besoin de réaliser cette tâche, il appelle la fonction. La programmation structurée s'appuie sur l'utilisation de ces fonctions afin d'obtenir un code modulaire et une approche "top-down". Les programmes ainsi développés sont plus performants et faciles à utiliser.

Une fonction est constituée d'un en-tête et d'un corps. L'en-tête contient le nom de la fonction, les paramètres et le type de valeur qu'elle va renvoyer. Le corps contient les déclarations de variables locales et les instructions qui seront exécutées à l'appel de la fonction. Les variables locales, déclarées dans la fonction, sont complètement indépendantes des autres variables du programme.

Q & R

Q Comment une fonction peut-elle renvoyer plusieurs valeurs ?

R Vous aurez souvent besoin de renvoyer plusieurs données à partir d'une seule fonction. Ce sujet est couvert par le Chapitre 18 qui vous donnera plus d'informations sur les fonctions.

Q Comment choisir un bon nom de fonction ?

R Le bon choix pour un nom de fonction est celui qui décrira le mieux ce que fait la fonction.

Q Quand les variables sont déclarées en début de programme, avant la fonction `main()`, on peut les utiliser à tout moment, sauf les variables locales qui sont utilisées dans les fonctions. Pourquoi ne pas faire toutes les déclarations avant `main()` ?

R Le Chapitre 12 vous donnera de plus amples informations concernant la portée des variables.

Q Comment peut-on employer la récurrence ?

R La fonction factorielle est un premier exemple d'utilisation de récurrence. Elle est souvent utilisée pour calculer des statistiques. La récurrence est une sorte de boucle qui, au contraire des autres boucles que vous étudierez dans le prochain chapitre, crée une nouvelle série de variables à chaque appel de la fonction.

Q La fonction `main()` doit-elle être la première à apparaître dans le programme ?

R Non. En langage C, la fonction `main()` est la première à être exécutée, mais elle peut se trouver n'importe où dans le fichier source. Les programmeurs la place en général en tête ou en fin du programme pour la localiser facilement.

Q Qu'est-ce qu'une fonction membre ?

R Une fonction membre est utilisée en langage C++ et Java. Elle fait partie d'une classe (structure spéciale employée en C++ et en Java).

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre. Essayez de comprendre les réponses fournies dans l'Annexe G avant de passer au chapitre suivant.

Quiz

1. Utiliserez-vous la programmation structurée pour écrire vos programmes ?
2. Comment fonctionne la programmation structurée ?
3. Les fonctions C sont-elles compatibles avec la programmation structurée ?
4. Quelle est la première ligne de la définition de fonction et quelles informations contient-elle ?

5. Combien de valeurs peut renvoyer une fonction ?
6. Si une fonction ne renvoie pas de valeur, quel type doit-elle avoir dans la déclaration ?
7. Quelle est la différence entre la définition et le prototype d'une fonction ?
8. Qu'est-ce qu'une variable locale ?
9. Quelle est la particularité des variables locales ?
10. Où doit être placée la fonction `main()` ?

Exercices

1. Écrivez l'en-tête de la fonction `fait le()` qui a trois arguments de type `char` et qui renvoie une valeur de type `float` au programme appelant.
2. Écrivez l'en-tête de la fonction `affiche un nombre()` qui a un seul argument de type `int` et qui ne renvoie rien au programme appelant.
3. Quel type de valeur renvoient les fonctions suivantes :
   a) `int affiche erreur(float err nbr);`.
   b) `long lit enreg(int rec nbr, int size);`.
4. **CHERCHEZ L'ERREUR** :

```
#include <stdio.h>
#include <stdlib.h>
void print_msg(void);
int main()
{
    print_msg("cela est un message à afficher");
   exit(EXIT_SUCCESS);
}
void print_msg(void)
{
    puts("cela est un message à afficher");
    return 0;
}
```

5. **CHERCHEZ L'ERREUR** : Où est l'erreur dans cette définition de fonction ?

```
int twice(int y);
{
    return (2 * y);
}
```

6. Transformez le Listing 5.4 pour qu'il n'ait besoin que d'une instruction `return` dans la fonction `larger of()`.

7. Écrivez une fonction qui reçoit deux nombres en arguments et qui renvoie la valeur correspondant au produit de ces deux nombres.
8. Écrivez une fonction qui reçoit deux nombres en arguments et qui divise le premier par le second si celui-ci est différent de zéro (utiliser une instruction `if`).
9. Écrivez une fonction qui appelle les fonctions des exercices 7 et 8.
10. Écrivez un programme qui utilise une fonction pour calculer la moyenne de cinq valeurs de type `float`, données par l'utilisateur.
11. Écrivez une fonction récurrente qui calcule le résultat de la valeur 3 à la puissance du nombre choisi par l'utilisateur. Par exemple, si le nombre 4 est tapé par l'utilisateur, le résultat sera 81.

6

Les instructions de contrôle

Le Chapitre 4 vous a donné quelques notions concernant le contrôle de l'exécution d'un programme avec l'ordre `if`. Cependant, vous aurez souvent besoin d'un contrôle plus sophistiqué qu'un simple test d'une condition vraie ou fausse. Ce chapitre vous donne trois nouvelles méthodes pour contrôler le flux de vos programmes. Aujourd'hui, vous allez apprendre à :

- Utiliser des tableaux simples
- Utiliser les boucles `for`, `while`, et `do while` pour exécuter des instructions plusieurs fois
- Imbriquer les instructions de contrôle dans un programme

Le Chapitre 13 complétera les informations de ce chapitre qui sont destinées à vous fournir des bases au sujet des structures de contrôle. Nous espérons ainsi permettre de commencer à écrire de vrais programmes.

Les tableaux

Avant d'aborder l'instruction `for`, nous allons vous donner quelques notions à propos des tableaux, qui seront traités plus en détail au Chapitre 8. L'instruction `for` est intimement liée à la notion de tableau. En langage C, on ne peut définir l'un sans expliquer l'autre.

Un *tableau* est un ensemble d'emplacements mémoire servant à stocker des données de même nom et auxquelles on accède par un *index* (adresse). L'index est représenté entre crochets [] à la suite du nom de la variable. Les tableaux devront être déclarés comme toutes les autres variables du langage C. La déclaration spécifie le type de donnée et la taille du tableau (c'est-à-dire, le nombre d'éléments qu'il contient). Voici la déclaration du tableau `data` de type `int`, qui contient 1000 éléments :

```
int data[1000]
```

Les différents éléments sont indexés de `data[0]` jusqu'à `data[999]`. Le premier élément du tableau est `data[0]`, et non `data[1]` comme on pourrait naturellement le penser.

Chaque élément est l'équivalent d'une variable entière normale. L'index peut être une variable, comme le montre l'exemple suivant :

```
int data[1000];
int compte;
compte = 100;
data[compte] = 12    /* l'équivalent de data[100] = 12 */
```



À ne pas faire

Déclarer des tableaux et des index trop grands : vous gaspillez la mémoire.

Oublier que, en C, l'index des tableaux commence à l'indice 0, et non 1.

Contrôle de l'exécution du programme

Par défaut, l'ordre d'exécution d'un programme C se fait de haut en bas (top-down). L'exécution commence avec la première instruction de la fonction `main()`, et se poursuit, instruction par instruction, jusqu'à la dernière. Le langage C contient de nombreuses instructions de contrôle qui permettent de modifier cet ordre d'exécution. Nous avons étudié l'ordre `if`, voici trois autres instructions que vous trouverez très utiles.

L'instruction *for*

L'instruction `for` permet d'exécuter un certain nombre de fois un bloc d'une ou plusieurs instructions. On l'appelle quelquefois la *boucle* `for`, parce que l'exécution du programme boucle sur ces instructions plus d'une fois. L'instruction `for` a la structure suivante :

```
for
(initial; condition; incrément)
    instruction(s)
```

`Initial`, *`condition`* et *`incrément`* sont des expressions ; *`instruction(s)`* représente une instruction simple ou composée. La boucle `for` fonctionne de la façon suivante :

1. L'expression `initial` est évaluée. Il s'agit en général d'une instruction d'affectation qui initialise une valeur particulière.
2. L'expression `condition` est évaluée. `condition` est souvent une expression de comparaison.
3. Si `condition` est fausse (valeur 0), l'instruction `for` se termine et l'exécution reprend à la première instruction qui suit `instruction(s)`.
4. Si `condition` est vraie(valeur différente de 0), les instructions de `instruction(s)` sont exécutées.
5. L'expression `incrément` est calculée et l'exécution reprend à l'étape 2.

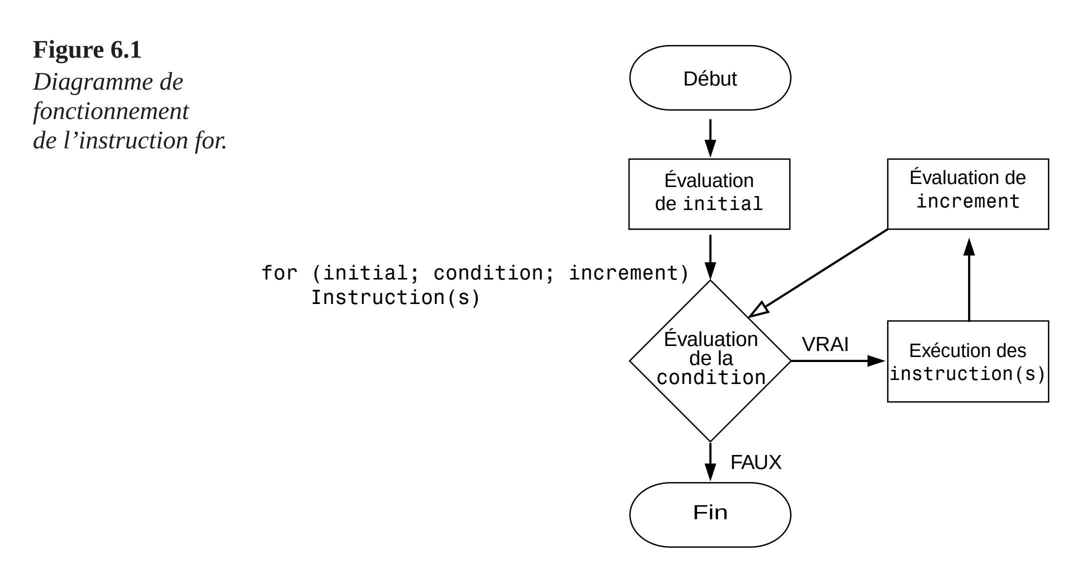


Figure 6.1
Diagramme de fonctionnement de l'instruction for.

Le Listing 6.1 présente un exemple simple de programme utilisant une boucle `for`. Ce programme affiche les nombres de 1 à 20. Le code est, bien sûr, beaucoup plus concis que si vous aviez utilisé une instruction `printf()` pour chaque valeur.

Listing 6.1 : L'instruction for

```
 1: /* Exemple simple d'utilisation d'une instruction for */
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4:
 5: int count;
 6:
 7: int main()
 8: {
 9:     /* Affichage des nombres de 1 à 20 */
10:
11:    for (count = 1; count <= 20; count++)
12:        printf("%d\n", count);
13:    exit(EXIT_SUCCESS);
14: }
```

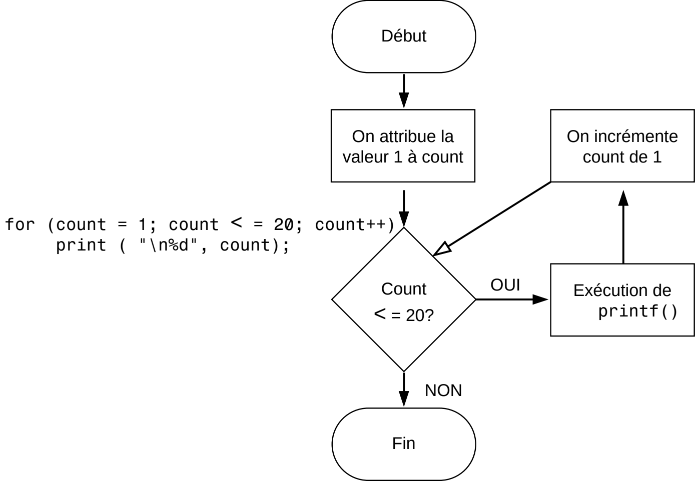
```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Analyse

La ligne 3 de ce programme appelle le traditionnel fichier d'entrées/sorties. La ligne 5 déclare une variable `count` de type `int` qui sera utilisée dans la boucle. Les lignes 11 et 12 constituent la boucle `for`. Quand l'exécution atteint l'instruction `for`, l'instruction initiale `count = 1` est exécutée : la variable `count` est initialisée pour la suite. La

deuxième étape est l'évaluation de la condition `count <= 20`. `count` étant égale à 1, la condition est vraie, l'instruction `printf()` est donc exécutée. L'étape suivante est le calcul de l'incrémentation `count++` qui ajoute 1 à la valeur de `count`. Le programme recommence alors la boucle et contrôle la condition de nouveau. Si elle est vraie, `printf()` sera exécutée de nouveau, la variable `count` sera incrémentée et la condition évaluée. Le programme va boucler sur ces instructions jusqu'à ce que la condition devienne fausse. Il sortira alors de la boucle pour continuer l'exécution ligne 13 (qui renvoie 0).

Figure 6.2
Voici comment opère la boucle for du Listing 6.1.



L'instruction `for` est souvent utilisée, comme dans l'exemple précédent, pour "compter" en incrémentant un compteur jusqu'à une valeur donnée. Vous pouvez bien entendu décrémenter un compteur de la même façon.

```
for (count = 100; count > 0; count --)
```

Le compteur peut être incrémenté d'une valeur différente de 1 :

```
for (count = 0; count < 1000; count +=5)
```

L'instruction `for` n'est pas contraignante. Vous pouvez omettre l'expression d'initialisation du "compteur" si la variable a été initialisée auparavant. Vous devez tout de même conserver le point-virgule :

```
count = 1;
for ( ; count < 1000; count++)
```

Cette expression d’initialisation peut être n’importe quelle expression C. Elle n’est exécutée qu’une fois au début de l’exécution de la boucle `for`. Voici un exemple :

```
count = 1;
for (printf("Nous allons trier le tableau") ; count < 1000; count++)
/* instructions de tri */
```

Vous pouvez aussi omettre l’expression d’incrémentation et mettre le compteur à jour dans les instructions de la boucle `for`. Voici un exemple de boucle qui affiche les nombres de 0 à 99 :

```
for (count = 0; count < 100;)
    printf("%d", count++);
```

L’expression de test qui termine la boucle peut être n’importe quelle expression C. Tant que cette expression est vraie (différente de 0), l’instruction `for` continue à s’exécuter. Vous pouvez utiliser les opérateurs logiques pour construire des expressions de test complexes. Par exemple, l’instruction `for` suivante affiche les éléments d’un tableau appelé `tableau[]`, jusqu’à ce qu’elle rencontre la valeur 0 ou la fin du tableau :

```
for (count = 0; count < 1000 && tableau[count] != 0; count++)
    printf("%d", tableau[count]);
```

Cette boucle `for` peut être simplifiée de la façon suivante :

```
for (count = 0; count < 1000 && tableau[count];)
    printf("%d", tableau[count++]);
```

Une boucle `for` peut avoir une instruction nulle, tout le travail étant fait dans l’instruction `for`. L’exemple suivant initialise les 1000 éléments d’un tableau à la valeur 50 :

```
for (count = 0; count < 1000; tableau[count++]=50)
;
```

Au Chapitre 4, nous avions annoncé que l’opérateur virgule était souvent utilisé dans les boucles `for`. En effet, vous pouvez créer une expression en séparant deux sous-expressions par une virgule. Les deux sous-expressions sont évaluées de gauche à droite, et l’expression prend la valeur de la sous-expression de droite. En utilisant la virgule, vous pouvez faire exécuter plusieurs fonctions par une seule instruction `for`.

Supposons que nous ayons deux tableaux de 1000 éléments `a[]` et `b[]`. On veut copier le contenu de `a[]` dans `b[]` dans un ordre inverse de telle sorte que `b[0]=a[999]`, `b[1]=a[998]`, etc.. Voici l’instruction `for` correspondante :

```
for (i = 0, j = 999; i < 1000; i++, j--)
    b[j] = a[i];
```

La virgule a permis d'initialiser les deux variables `i` et `j`, puis de les incrémenter à chaque boucle.

Syntaxe de la commande *for*

```
for (initial; condition; incrément)
instruction(s)
```

`initial` est une expression du langage C, généralement une instruction d'affectation qui initialise une variable.

`condition` est une expression C, habituellement une comparaison. Quand `condition` est fausse (valeur 0), l'instruction `for` se termine et l'exécution se poursuit avec la première instruction qui suit `instruction(s)`. Dans le cas contraire, les instructions de `instruction(s)` sont exécutées.

`incrément` est une expression C qui, en règle générale, `incrémente` la valeur de la variable initialisée dans l'expression `initial`.

`instruction(s)` représente les instructions exécutées aussi longtemps que la condition reste vraie.

L'instruction `for` est une instruction qui boucle. L'expression `initial` est exécutée la première, suivie de la `condition`. Si celle-ci est vraie, les instructions sont exécutées et l'expression `incrément` est calculée. La condition est alors contrôlée de nouveau et l'exécution se poursuit jusqu'à ce que la condition devienne fausse.

Exemple 1

```
/* Affiche la valeur de x en comptant de 0 à 9 */
int x;
for (x=0; x<10; x++)
    printf("\nLa valeur de x est %d", x);
```

Exemple 2

```
/* Demande une valeur à l'utilisateur jusqu'à l'obtention de 99 */
int nbr = 0;
for ( ; nbr != 99;)
    scanf("%d", &nbr);
```

Exemple 3

```
/* Permet à l'utilisateur d'entrer 10 valeurs entières. */
/* Les valeurs sont stockées dans un tableau appelé valeur. */
/* La boucle est interrompue si l'utilisateur tape 99. */
```

```
int valeur[10];
int ctr, nbr=0;
for (ctr = 0; ctr < 10 && nbr != 99; ctr++)
{
    puts("Entrez un nombre ou 99 pour sortir");
    scanf("%d", &nbr);
    valeur[ctr] = nbr;
}
```

Instructions *for* imbriquées

On peut exécuter une instruction `for` à l'intérieur d'une autre instruction `for`. En imbriquant des instructions `for`, on peut réaliser une programmation très complexe.

Listing 6.2 : Instructions for imbriquées

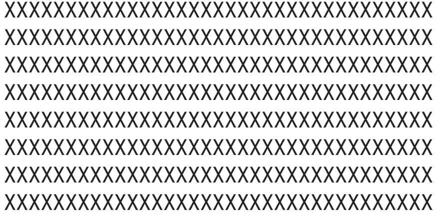
```
/* Exemple de deux instructions for imbriquées */
#include <stdio.h>
#include <stdlib.h>

void boite(int ligne, int colonne);

int main()
{
    boite(8, 35);
   exit(EXIT_SUCCESS);
}

void boite(int ligne, int colonne)
{
   int col;
   for( ; ligne > 0; ligne--)
   {
      for(col = colonne; col > 0; col--)
         printf("X");

      printf("\n");
   }
}
```



```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Analyse

Le travail le plus important de ce programme est réalisé à la ligne 18. Quand vous l'exécutez, 280 "X" s'affichent, formant un tableau de 8 lignes et 35 colonnes. Le programme ne contient qu'une seule commande d'affichage du caractère "X", mais elle se trouve à l'intérieur de deux boucles imbriquées.

La ligne 5 de ce programme source contient le prototype de la fonction `boîte()`. Cette fonction utilise deux variables de type `int`, `ligne` et `colonne`, qui représentent les dimensions du tableau de "X". La fonction principale `main()` appelle la fonction `boîte()` en ligne 9 et lui transmet la valeur 8 pour les lignes et 35 pour les colonnes.

Étudions la fonction `boîte()`. Deux choses vous paraissent certainement étranges : pour quelle raison a-t-on déclaré une variable locale `col`, et pourquoi a-t-on utilisé une seconde fois la fonction `printf()` en ligne 21 ? Ces deux points vont être éclaircis par l'étude des deux boucles `for`.

La première commence en ligne 16. Elle ne contient pas de partie d'initialisation, car la valeur initiale de `ligne` a été transmise à la fonction. La partie `condition` indique que cette boucle s'exécutera jusqu'à ce que `ligne` soit égale à 0. À la première exécution de cette boucle, `ligne` est égale à 8. L'exécution du programme se poursuit en ligne 18.

La ligne 18 contient la seconde boucle `for`. Le paramètre transmis est `colonne`, que l'on copie dans la variable `col` de type `int`. La valeur initiale de `col` est donc 35 et `colonne` va conserver cette valeur inchangée. La variable `col` est supérieure à 0, la ligne 19 est donc exécutée : on affiche un "X" sur l'écran. La valeur de `col` est diminuée de 1 et la boucle continue. Quand `col` atteint la valeur 0, la boucle se termine et la ligne 21 prend le contrôle. Cette ligne envoie vers l'écran un retour à la ligne. C'est maintenant la dernière étape de la première boucle : La partie `incrément` s'exécute en diminuant la valeur de `ligne` de 1, ce qui lui donne la valeur 7. Le contrôle est donné de nouveau à la ligne 18. Remarquez bien que la dernière valeur de `col` était 0 ; si on avait utilisé la variable `colonne` le test de condition serait déjà faux et on n'aurait imprimé qu'une seule ligne.



À faire

Ne pas oublier le point-virgule dans une boucle `for`, avec une instruction nulle. Pour plus de clarté, mettez-le sur une ligne séparée, ou à la fin de l'instruction `for` en l'isolant par un espace :

```
for (count = 0; count < 1000; tableau[count] = 50) ;
```

À ne pas faire

Écrire une instruction `for` trop chargée. Bien que l'on puisse utiliser des virgules, il est souvent plus simple de mettre certaines fonctionnalités dans le corps de la boucle.

L'instruction *while*

L'instruction `while`, que l'on peut appeler la *boucle* `while`, exécute un bloc d'instructions tant qu'une condition reste vraie.

```
while (condition)
    instruction(s)
```

La `condition` est une expression C et `instruction(s)` représente une instruction simple ou composée. La boucle `while` fonctionne de la façon suivante :

1. La `condition` est évaluée.
2. Si `condition` est fausse (valeur 0), l'instruction `while` se termine et l'exécution se poursuit avec l'instruction qui suit immédiatement `instruction(s)`.
3. Si `condition` est vraie (valeur différente de 0), les instructions de `instruction(s)` sont exécutées.
4. L'exécution reprend à l'étape 1.

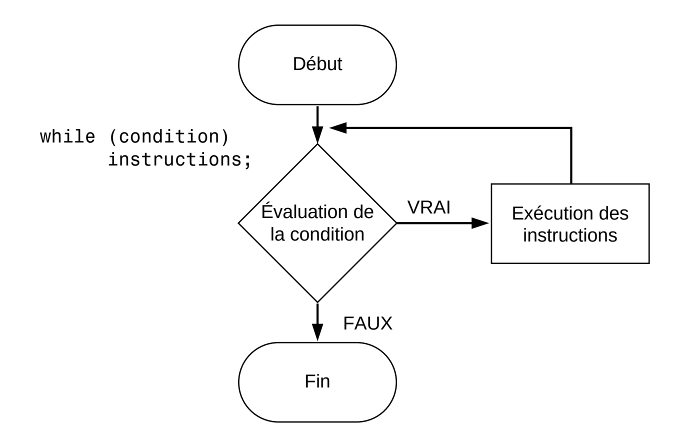


Figure 6.3
Diagramme de fonctionnement d'une instruction while.

Le Listing 6.3 décrit un programme simple utilisant une instruction `while` pour imprimer des nombres de 1 à 20.

Listing 6.3 : L'instruction while

```
/* Exemple d'exécution d'une instruction while simple */
#include <stdio.h>
#include <stdlib.h>

int count;

```

```
 7: int main()
 8: {
 9:     /* Affiche les nombres de 1 à 20 */
10:
11:    count = 1;
12:
13:    while (count <= 20)
14:    {
15:        printf("%d\n", count);
16:        count++;
17:    }
18:    exit(EXIT_SUCCESS);
19: }
```

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
```

Analyse

Comparez le Listing 6.3 avec le Listing 6.1 qui exécutait la même tâche avec une instruction `for`. En ligne 11, la variable `count` est initialisée à 1. L'instruction `while` ne contient pas de partie d'initialisation, qui doit donc être faite avant. L'instruction `while` se trouve en ligne 13 et contient la même `condition` que la boucle `for` du Listing 6.1 : `count <= 20`. Dans la boucle `while`, c'est la ligne 16 qui est chargée d'incrémenter la variable `count`. Si vous aviez oublié de coder cette ligne, le programme n'aurait pas su quand s'arrêter, car `count` aurait conservé la valeur 1 qui est toujours inférieure à 20.

Une instruction `while` est une instruction `for` sans les deux parties `initialisation` et `incrément`.

```
for ( ; condition ;)
```

est donc l'équivalent de :

```
while (condition)
```

Tout ce que vous pouvez faire avec `for` peut être fait avec `while`. Si vous utilisez l'instruction `while`, vous devez initialiser vos variables avant, et le "compteur" doit être mis à jour dans la partie `instruction(s)`.

Quand les parties `initialisation` et `incrément` sont nécessaires, les programmeurs expérimentés préfèrent utiliser l'instruction `for`. Les trois parties `initialisation`, `condition`, et `incrément` étant réunies sur la même ligne, le code est plus facile à lire et à modifier.

Syntaxe de la commande *while*

```
while (condition)
    instruction(s)
```

`condition` est une expression du langage C. En général, c'est une expression de comparaison. Quand cette `condition` est fausse (valeur 0), l'instruction `while` se termine et l'exécution se poursuit avec l'instruction qui suit *`instruction(s)`*. Sinon, les *`instruction(s)`* sont exécutées.

`instruction(s)` représente des instructions qui sont exécutées tant que `condition` reste vraie.

`while` est une instruction C qui boucle. Elle permet de répéter l'exécution d'une instruction, ou d'un bloc d'instructions, tant qu'une condition reste vraie (valeur différente de 0). Si la condition est fausse à la première exécution de l'instruction `while`, les `instruction(s)` ne sont jamais exécutées.

Exemple 1

```
int x = 0;
while (x < 10)
{
    printf("La valeur de x est %d\n", x);
    x++;
}
```

Exemple 2

```
/* saisie des nombres entrés par l'utilisateur et sortie si supérieur à 99 */
int nbr = 0;
while (nbr <= 99)
    scanf("%d", &nbr);
```

Exemple 3

```
/* L'utilisateur peut entrer jusqu'à 10 valeurs entières */
/* Ces valeurs sont stockées dans un tableau appelé valeur */
/* la boucle se termine si l'utilisateur rentre 99 */
int valeur[10];
int ctr = 0;
int nbr;
while (ctr < 10 && nbr != 99)
{
    puts("Entrez un nombre ou 99 pour sortir ");
    scanf("%d", &nbr);
    valeur[ctr] = nbr;
    ctr++;
}
```

Instructions *while* imbriquées

On peut imbriquer des instructions `while` comme avec `for` ou `if`. Le Listing 6.4 vous en montre un exemple qui ne représente pas le meilleur usage de `while`, mais qui vous propose quelques idées nouvelles.

Listing 6.4 : Instructions while imbriquées

```
/* Exemple d'instructions while imbriquées */
#include <stdio.h>
#include <stdlib.h>

int tableau[5];

int main()
{
     int ctr = 0,
    nbr = 0;

    printf("Ce programme vous demande d'entrer 5 nombres,\n");
    printf("chacun compris entre 1 et 10\n");

    while (ctr < 5)
    {
        nbr = 0;
        while (nbr < 1 || nbr > 10)
        {
            printf("\nEntrez le nombre numéro %d sur 5 : ", ctr + 1);
            scanf("%d", &nbr);
        }
```

Listing 6.4 : Instructions while imbriquées *(suite)*

```
23:
24:         tableau[ctr] = nbr;
25:         ctr++;
26:     }
27:
28:     for(ctr = 0; ctr < 5; ctr++)
29:         printf("La valeur %d est %d\n", ctr + 1, tableau[ctr]);
30:     exit(EXIT_SUCCESS);
31: }
```

```
Ce programme vous demande d'entrer 5 nombres
chacun compris entre 1 et 10

Entrez le nombre numéro 1 sur 5 : 3

Entrez le nombre numéro 2 sur 5 : 6

Entrez le nombre numéro 3 sur 5 : 3

Entrez le nombre numéro 4 sur 5 : 9

Entrez le nombre numéro 5 sur 5 : 2

La valeur 1 est 3
La valeur 2 est 6
La valeur 3 est 3
La valeur 4 est 9
La valeur 5 est 2
```

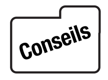
Analyse

Vous retrouvez en ligne 1 le commentaire de description du programme ; la ligne 3 appelle le traditionnel fichier d'en-tête. La ligne 5 déclare le tableau dans lequel on pourra stocker cinq valeurs entières. La fonction `main()` contient deux variables locales, `ctr` et `nbr`, qui sont déclarées et initialisées en lignes 9 et 10. Remarquez l'utilisation de la virgule qui est une pratique courante des programmeurs. Cela permet de déclarer une variable par ligne, sans avoir à répéter `int`. Les lignes 12 et 13 affichent pour l'utilisateur ce que le programme lui demande de faire. La première boucle `while`, lignes 15 à 26, contient une boucle `while` imbriquée en lignes 18 à 22. Elle va s'exécuter tant que la variable `ctr` sera inférieure à 5. Cette première boucle initialise `nbr` à 0 à la ligne 17, puis la boucle imbriquée récupère le nombre entré par l'utilisateur pour la variable `nbr`. La ligne 24 stocke ce nombre dans le tableau et la valeur de `ctr` est incrémentée en ligne 25. Ensuite, la boucle principale recommence.

La boucle intérieure est un bon exemple d'utilisation de `while`. Les nombres valides sont les nombres de 1 à 10 ; tant que l'utilisateur choisit un nombre dans cet intervalle, l'exécution n'a pas de raison de s'interrompre. Ce sont les lignes 18 à 22 qui contrôlent le nombre

entré par l'utilisateur : l'instruction `while` va envoyer le même message à l'utilisateur tant que ce nombre, `nbr`, sera inférieur à 1 ou supérieur à 10.

Enfin, les lignes 28 et 29 impriment tous les nombres qui ont été stockés dans le tableau.



À faire

Utilisez l'instruction `for` plutôt que `while` si vous avez besoin d'initialiser et d'incrémenter dans la boucle. L'instruction `for` réunit l'initialisation, la condition, et l'incrémentation sur la même ligne.

À ne pas faire

Coder des instructions du genre `while (x)` lorsque ce n'est pas nécessaire.

Écrivez plutôt `while (x != 0)` ; les deux instructions sont correctes, mais la seconde sera plus facile à corriger en cas de problème.

La boucle *do-while*

La troisième boucle est la boucle `do while` qui exécute un bloc d'instructions tant qu'une condition reste vraie. La différence entre la boucle `do while` et les deux boucles précédentes est que le test de la condition s'effectue à la fin de la boucle. Pour les deux autres, le test se fait au début.

L'instruction `do while` a la structure suivante :

```
do
    instructions
while (condition);
```

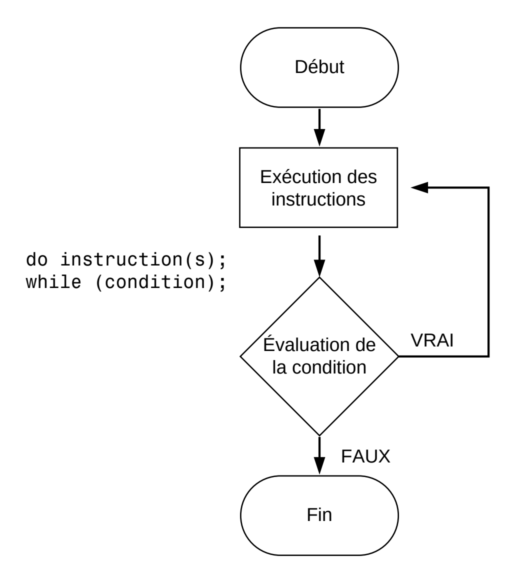
`condition` est une expression du langage C, et `instruction(s)` représente une instruction simple ou composée. Quand une instruction `do while` est rencontrée pendant l'exécution du programme, voici ce qui se passe :

1. Les instructions de `instruction(s)` sont exécutées.
2. `condition` est testée. Si elle est vraie, l'exécution reprend à l'étape 1. Si elle est fausse, la boucle se termine.

Le test de condition étant réalisé à la fin, les instructions qui font partie de la boucle `do while` sont toujours exécutées au moins une fois. Au contraire, dans les boucles `for` et `do while`, si le test est faux dès le début, les instructions ne sont jamais exécutées.

La boucle `do while` est moins souvent utilisée que `for` ou `while`. Son intérêt réside surtout dans le fait que les instructions sont exécutées au moins une fois.

Figure 6.4
Diagramme de fonctionnement de la boucle do-while.



Listing 6.5 : La boucle do while

```
/* Exemple simple d'utilisation de l'instruction do-while */
#include <stdio.h>
#include <stdlib.h>

int choix_menu(void);

int main()
{
    int choix;

   choix = choix_menu();

   printf("Vous avez choisi l'option %d\n du menu", choix);
   exit(EXIT_SUCCESS);
}

int choix_menu(void)
{
   int selection = 0;

   do
   {
       printf("\n");
       printf("1 - Ajouter un enregistrement\n");
       printf("2 - Changer un enregistrement\n");
       printf("3 - Effacer un enregistrement\n");
       printf("4 - Sortie\n");
       printf("\n");
       printf("Entrez votre choix :");
```

```
30:
31:        scanf("%d", &selection);
32:
33:    }while (selection < 1 || selection > 4);
34:
35:    return selection;
36: }
```

```
1 - Ajouter un enregistrement
2 - Changer un enregistrement
3 - Effacer un enregistrement
4 - Sortir

Entrez votre choix : 8

1 - Ajouter un enregistrement
2 - Changer un enregistrement
3 - Effacer un enregistrement
4 - Sortir

Entrez votre choix : 4
Vous avez choisi l'option 4 du menu
```

Analyse

Ce programme propose un menu comprenant quatre options. L'utilisateur en choisit une et le programme lui affiche le numéro sélectionné. Ce concept sera largement repris par la suite avec des programmes plus complexes. La fonction principale `main()` (lignes 7 à 14) ne comporte aucune nouveauté.



La fonction `main()` *aurait pu être écrite sur une seule ligne :*

printf("Vous avez choisi l'option %d du menu", choix_menu());

Si vous aviez besoin de prolonger ce programme en associant une tâche à chaque sélection, vous auriez besoin de la valeur renvoyée par `choix menu`*. Il est donc préférable de l'affecter à une variable (choix).*

Le code de la fonction `choix menu` se trouve aux lignes 17 à 36. Cette fonction permet d'afficher le menu à l'écran (lignes 23 à 29) et de récupérer le numéro choisi par l'utilisateur. La boucle `do while` a été choisie, car le menu doit être affiché au moins une fois pour connaître la décision de l'utilisateur. Dans ce cas, le menu s'affiche jusqu'à ce qu'un numéro valide soit entré au clavier. La ligne 33 contient la partie `while` de la boucle et contrôle la valeur sélectionnée (`sélection`). Si cette valeur n'est pas un nombre entier compris entre 1 et 4, le menu réapparaît et un message demande à l'utilisateur d'effectuer un autre choix. Si la valeur est correcte, l'exécution se poursuit ligne 35 avec le stockage de cette valeur dans la variable `selection`.

Syntaxe de la commande *do-while*

```
do
{
    instruction(s)
}while (condition);
```

`condition` est une expression du langage C, en règle générale une comparaison. Quand la `condition` est fausse (zéro), l'instruction `while` se termine et l'exécution se poursuit avec l'instruction qui suit celle de `while`. Sinon le programme reprend au niveau du `do` et les instructions de `instruction(s)` sont exécutées.

`instruction(s)` représente une ou plusieurs instructions exécutées une première fois au démarrage de la boucle, puis tant que la condition reste vraie.

Une instruction `do while` est une instruction C qui boucle. Elle permet d'exécuter une instruction ou un bloc d'instructions aussi longtemps qu'une condition reste vraie. Contrairement à `while`, la boucle `do while` s'exécute au moins une fois.

Exemple 1

```
/* Le message est affiché même si la condition est fausse ! */
int x = 10;
do
{
    printf("La valeur de x est %d\n", x);
}while (x != 10);
```

Exemple 2

```
/* Lit des nombres jusqu'à ce que le nombre entré soit supérieur à 99 */
int nbr;
do
{
    scanf("%d", &nbr);
}while (nbr <= 99);
```

Exemple 3

```
/* Permet à l'utilisateur d'entrer dix valeurs entières */
/* Ces valeurs sont stockées dans un tableau appelé valeur */
/* On sort de la boucle si l'utilisateur entre 99 */
int valeur[10];
int ctr = 0;
int nbr;
do
{
    puts("Entrez un nombre, ou 99 pour sortir");
    scanf("%d", &nbr);
    ctr++;
}while (ctr < 10 && nbr != 99);
```

Les boucles imbriquées

Le terme *boucle imbriquée* fait référence à une boucle située à l'intérieur d'une autre boucle. Le langage C n'est pas contraignant sur l'utilisation de telles boucles. La seule contrainte est que la boucle intérieure doit être entièrement contenue dans la boucle extérieure. Voici un exemple de boucle qui "déborde" à ne pas suivre :

```
for (count = 1; count < 100; count++)
{
    do
    {
    /* la boucle do-while */
} /* fin de la boucle for */
   } while (x != 0);
```

Voici le même exemple corrigé :

```
for (count = 1; count < 100; count++)
{
    do
    {
    /* la boucle do-while */
    } while (x != 0);
} /* fin de la boucle for */
```

Rappelez-vous, lorsque vous utiliserez des boucles imbriquées, que des changements réalisés dans la boucle intérieure peuvent affecter la boucle extérieure. Les variables de la boucle intérieure peuvent cependant être indépendantes. Dans notre exemple ce n'est pas le cas, mais dans l'exemple précédent, la boucle intérieure agissant sur la valeur de `count`, modifiait le nombre d'exécutions de la boucle `for`.

Prenez l'habitude de décaler d'une tabulation chaque niveau de boucle pour les différencier facilement.



À faire

Inclure entièrement une boucle imbriquée dans la boucle extérieure. Il ne doit pas y avoir de recouvrement.

Utiliser la boucle `do while` quand il faut exécuter les instructions au moins une fois.

Résumé

Vous avez maintenant le matériel nécessaire pour commencer à coder vos programmes.

Le langage C possède trois instructions de boucle pour contrôler l'exécution des programmes : `for`, `while`, et `do while`. Chacune de ces instructions permet d'exécuter zéro, une ou plusieurs fois une instruction ou un bloc d'instructions, en fonction du test d'une condition. En programmation, de nombreuses tâches peuvent être réalisées avec des boucles.

Dans le principe, ces trois boucles peuvent réaliser les mêmes tâches, mais elles sont différentes. L'instruction `for` permet, en une seule ligne de code, d'initialiser, d'évaluer, et d'incrémenter. L'instruction `while` s 'exécute tant que la condition reste vraie. Enfin, l'instruction `do while` s'exécute une fois, puis de nouveau jusqu'à ce que la condition soit fausse.

Le terme de boucle imbriquée désigne une boucle complètement incluse dans une autre boucle. Le langage C permet d'imbriquer toutes ses commandes. Nous avons vu comment imbriquer des ordres `if` dans le Chapitre 4.

Q & R

Q Comment choisir entre `for`, `while`, et `do while` ?

R Si vous étudiez la syntaxe de ces trois boucles, vous remarquez qu'elles servent toutes à résoudre un problème de boucle tout en ayant chacune une particularité. Si votre boucle a besoin de l'initialisation et de l'incrémentation d'une variable, l'instruction `for` sera plus appropriée. Si la condition est importante et que le nombre d'exécutions de la boucle importe peu, alors le choix de `while` est judicieux. Si les instructions ont besoin d'être exécutées au moins une fois, `do while` sera le meilleur choix.

Q Combien de niveaux de boucles puis-je imbriquer ?

R Vous pouvez imbriquer autant de niveaux de boucles que vous le voulez. Toutefois, si vous avez besoin d'imbriquer plus de deux niveaux de boucles dans votre programme, essayez d'utiliser une fonction. Cela diminuera le nombre d'accolades nécessaires, et une fonction sera peut-être plus facile à coder et à corriger.

Q Puis-je imbriquer des instructions de boucle différentes ?

R Vous pouvez imbriquer une boucle `if`, `for`, `while`, `do while` dans n'importe quelle autre commande. Vous en aurez souvent besoin au cours de vos programmations.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre. Essayez de comprendre les réponses fournies dans l'Annexe G avant de passer au chapitre suivant.

Quiz

1. Quelle est la valeur d'index du premier élément d'un tableau ?
2. Quelle est la différence entre une instruction `for` et une instruction `while` ?
3. Quelle est la différence entre une instruction `while` et une instruction `do while` ?
4. Une instruction `while` peut-elle donner le même résultat qu'une instruction `for` ?
5. De quoi faut-il se rappeler à propos des instructions imbriquées ?
6. Peut-on imbriquer une instruction `while` dans une instruction `do while` ?
7. Quelles sont les quatre parties d'une instruction `for` ?
8. Quelles sont les deux parties d'une instruction `while` ?
9. Quelles sont les deux parties d'une instruction `do…while` ?

Exercices

1. Écrivez la déclaration correspondant à un tableau qui contiendra 50 valeurs de type `long`.
2. Écrivez l'instruction qui attribue la valeur 123,456 au cinquantième élément du tableau de l'exercice 1.
3. Quelle est la valeur de `x` après l'exécution de l'instruction suivante ?

   ```
   for (x = 0; x < 100, x++);
   ```

4. Quelle est la valeur de `ctr` après l'exécution de l'instruction suivante ?

   ```
   for (ctr = 0; ctr < 10; +=3);
   ```

5. Combien de caractères X la boucle `for` suivante affiche-t-elle ?

   ```
   for (x = 0; x < 10; x++)
       for (y = 5; y > 0; y--)
       puts("X");
   ```

6. Écrivez une instruction `for` pour compter de 1 à 100 de 3 en 3.

7. Écrivez une instruction `while` pour compter de 1 à 100 de 3 en 3.
8. Écrivez une instruction `do while` pour compter de 1 à 100 de 3 en 3.
9. **CHERCHEZ L'ERREUR :**

```
record = 0;
while (record < 100)
{
    printf("Enregistrement %d\n", record);
    printf("Prochain nombre ...\n");
}
```

10. **CHERCHEZ L'ERREUR** (Ce n'est pas MAXVALUES !)

```
for (counter = 1; counter < MAXVALUES; counter++);
    printf("Counter = %d\n", counter);
```

7

Les principes de base des entrées/sorties

La plupart des programmes que vous allez écrire auront besoin d'afficher des informations à l'écran, ou de lire des données entrées au clavier. Aujourd'hui, vous allez apprendre à :

- Afficher des informations à l'écran avec les fonctions de bibliothèque `printf()` et `puts()`
- Mettre en forme les messages envoyés vers l'écran
- Lire les données entrées au clavier avec la fonction de bibliothèque `scanf()`

Afficher des informations à l'écran

Les deux fonctions de bibliothèque les plus souvent utilisées pour afficher des informations à l'écran sont `printf()` et `puts()`.

La fonction *printf()*

Nous avons vu de nombreux exemples d'utilisation de `printf()`, mais vous n'en connaissez pas encore le mode de fonctionnement. La fonction `printf()`, qui fait partie de la bibliothèque standard du C, est la plus polyvalente pour afficher des informations à l'écran.

Afficher un texte sur l'écran est très simple : il suffit d'appeler la fonction `printf()` et de lui passer le message entre guillemets(`" "`). L'instruction suivante affiche le message `une erreur s'est produite !`

```
printf("une erreur s'est produite !");
```

Pour introduire la valeur d'une variable dans votre message, c'est un peu plus compliqué. Supposons, par exemple, que vous vouliez afficher la valeur de la variable `x` avec un texte descriptif, et commencer le message sur une nouvelle ligne. Il faudra utiliser l'instruction suivante :

```
printf("La valeur de x est %d\n", x);
```

Si la valeur de `x` est 12, le résultat à l'écran de l'instruction précédente sera :

```
La valeur de x est 12
```

Dans notre exemple, nous avons transmis deux arguments à `printf()`. Le premier est la *chaîne format* qui se trouve entre guillemets, le second, le nom de la variable qui contient la valeur à afficher.

Les chaînes format de *print()*

Comme son nom l'indique, la chaîne format de `printf()` indique les spécifications de format pour la chaîne à émettre. Voici les trois éléments que l'on peut introduire dans une chaîne format :

- Un texte simple qui sera affiché tel quel. "`La valeur de x est`" représente la partie texte de l'exemple précédent.
- Un ordre de contrôle qui commence par un antislash (`\`) suivi d'un caractère. Dans notre exemple, `\n` est l'ordre de contrôle. C'est le caractère de *retour à la ligne* ; il signifie littéralement "se placer au début de la prochaine ligne".

- Une *spécification de conversion* qui est représentée par le signe de pourcentage (`%`) suivi d'un caractère. Celle de notre exemple est `%d`. Elle indique à la fonction `printf()` comment interpréter les variables que l'on veut afficher. `%d` signifie pour `printf()` que la variable `x` est un entier décimal signé.

Tableau 7.1 : Ordres de contrôle le plus souvent utilisés

Ordre	*Signification*
`\a`	Sonnerie
`\b`	Retour arrière
`\n`	Retour à la ligne
`\t`	Tabulation horizontale
`\\`	Backslash (antislash \)

Les ordres de contrôle de la chaîne format permettent de contrôler l'emplacement final du message en déplaçant le curseur. Ils permettent aussi d'afficher des caractères qui ont ordinairement une signification particulière pour `printf()`. Pour imprimer le caractère (`\`) par exemple, il faut en introduire deux (\\) dans la chaîne format. Le premier indique à `printf()` que le second devra être interprété comme un simple caractère, et non comme le début d'un ordre de contrôle. En règle générale, le caractère (`\`) demande à `printf()` d'interpréter le caractère qui suit d'une façon particulière. En voici quelques exemples :

Ordre	*Signification*
`n`	Le caractère *n*
`\n`	Retour à la ligne
`\"`	Le guillemet
`"`	Début ou fin d'une chaîne

Le Tableau 7.1 contient les ordres de contrôle les plus utilisés. Vous en trouverez la liste complète dans le Chapitre 15.

Listing 7.1 : Utilisation des ordres de contrôle avec printf()

```
/* Ce programme contient des ordres de contrôle
   souvent utilisés */
#include <stdio.h>
#include <stdlib.h>
#define QUIT  3

```

Listing 7.1 : Utilisation des ordres de contrôle avec printf() ***(suite)***

```
int  choix_menu(void);
void affiche(void);

int main()
{
   int choix = 0;

   while(choix != QUIT)
   {
    choix = choix_menu();

    if(choix == 1)
       printf("\nL'ordinateur va biper\a\a\a");
    else
    {
       if(choix == 2)
          affiche();
    }
   }
   printf("Vous avez choisi de sortir!\n");
   exit(EXIT_FAILURE);
}

int choix_menu(void)
{
   int selection = 0;

   do
   {
    printf("\n");
    printf("\n1 - Bip ordinateur");
    printf("\n2 - Affichage ");
    printf("\n3 - Sortir");
    printf("\n");
    printf("\nEntrez votre choix :");

    scanf("%d", &selection);

   }while (selection < 1 || selection > 3);

   return selection;
}

void affiche(void)
{
   printf("\nExemple d'affichage");
   printf("\n\nOrdre\tSignification");
   printf("\n======\t==============");
   printf("\n\\a\t\tsonnerie ");
   printf("\n\\b\t\tretour arrière");
   printf("\n...\t\t...");
}
```



```
1 - Bip ordinateur
2 - Affichage
3 - Sortir
```

```
Entrez votre choix : 1

L'ordinateur va biper

1 - Bip ordinateur
2 - Affichage
3 - Sortir

Entrez votre choix : 2

Exemple d'affichage
Ordre               Signification
=====               ==============
 \a                 sonnerie
 \b                 Retour arrière

1 - Bip ordinateur
2 - Affichage
3 - Sortir

Entrez votre choix : 3
Vous avez choisi de sortir !
```

Analyse

Le fichier en-tête stdio.h est inclus ligne 2 pour pouvoir utiliser la fonction `printf()`. La ligne 5 définit la constante `QUIT` et les lignes 7 et 8 le prototype des deux fonctions : `affi che()` et `choix menu()`. La définition de `choix menu` se trouve aux lignes 30 à 48. Cette fonction est analogue à la fonction menu du Listing 6.5. Les lignes 36 à 41 contiennent les appels de la fonction `printf()` avec les ordres de retour à la ligne. La ligne 36 pourrait être supprimée en transformant la ligne 37 de cette façon :

```
printf("\n\n1 - Bip ordinateur");
```

Examinons la fonction `main()`. Une boucle `while` commence en ligne 14 et s'exécute tant que l'utilisateur ne choisit pas de sortir (`choix` différent de `QUIT`). `QUIT` étant une constante, elle aurait pu être remplacée par la valeur 3. Quoi qu'il en soit, le programme y a gagné en clarté. La ligne 16 lit la variable `choix` qui est analysée par l'instruction `if` aux lignes 18 à 24. Si le choix de l'utilisateur est 1, la ligne 19 provoque un retour à la ligne, affiche le message et fait biper trois fois l'ordinateur. Si le choix est 2, la ligne 23 appelle la fonction `affiche()`.

La définition de la fonction `affiche()` se trouve aux lignes 50 à 58. Cette fonction facilite l'affichage à l'écran d'un texte mis en forme avec des retours à la ligne. Les lignes 53 à 57 utilisent l'ordre de contrôle `\t` (tabulation) pour aligner les colonnes du tableau. Pour comprendre les lignes 55 et 56, il faut analyser la chaîne de gauche à droite. La ligne 55 provoque un retour à la ligne (`\n`), affiche un antislash (`\`) suivi du caractère `a`, puis

déplace le curseur de deux tabulations (`\t \t`) et, enfin, affiche le texte `sonnerie`. La ligne 56 suit le même format.

Ce programme affiche les deux premières lignes du Tableau 7.1. Dans l'exercice 9 de ce chapitre, vous devrez compléter ce programme pour afficher le tableau entier.

Les spécifications de conversion de *printf()*

La chaîne format contient des spécifications de conversion qui doivent correspondre, en nombre et en type, aux arguments. Cela signifie que si vous voulez afficher une variable de type *entier décimal signé* (types `int` et `long`), vous devez inclure dans la chaîne `%d`. Pour un *entier décimal non signé* (types `unsigned int` et `unsigned long`), vous devez inclure `%u`. Enfin, pour une variable *à virgule flottante* (types `float` ou `double`), il faut utiliser `%f`.

Tableau 7.2 : Spécifications de conversion les plus fréquentes

Conversion	*Signification*	*Types convertis*
`%c`	Un seul caractère	`char`
`%d`	Entier décimal signé	`int, short,`
`%ld`	Entier décimal signé long	`long`
`%f`	Nombre à virgule flottante	`float, double`
`%s`	Chaîne de caractères	`tableaux char`
`%u`	Entier décimal non signé	`unsigned int, unsigned short`
`%lu`	Entier décimal non signé long	`unsigned long`

Le texte contenu dans une chaîne format, c'est-à-dire tout ce qui n'est pas ordre de contrôle ou spécification de conversion, est affiché de façon littérale en incluant tous les espaces.

L'instruction `printf()` n'est pas limitée quant au nombre de variables qu'elle peut afficher. La chaîne format doit cependant contenir une spécification de conversion pour chacune de ces variables. La correspondance conversion-variable se fait de gauche à droite. Par exemple, si vous écrivez :

```
printf("taux = %f, montant = %d", taux, montant);
```

la spécification de conversion `%f` sera remplacée par la variable `taux`, et `%d` sera remplacée par la variable `montant`. Si la chaîne contient plus de variables que de spécifications de

conversion, les variables supplémentaires n'apparaissent pas. S'il y a plus de spécifications de conversion que de variables, la fonction `printf()` affiche n'importe quoi.

Il n'y a pas de restriction sur ce que vous pouvez afficher avec la fonction `printf()`. Un argument peut être une expression du langage C. Pour afficher la somme de `x` et `y`, vous pouvez écrire, par exemple :

```
z = x + y;
printf("%d", z);
```

ou

```
printf("%d", x + y);
```

Un programme qui utilise une fonction `printf()` doit obligatoirement contenir le fichier en-tête stdio.h.

Listing 7.2 : Utilisation de la fonction printf() pour afficher des valeurs numériques

```
/*   Utilisation de printf() pour afficher des valeurs numériques.*/
#include <stdio.h>
#include <stdlib.h>

int a = 2, b = 10, c = 50;
float f = 1.05, g = 25.5, h = -0.1;

int main()
{
   printf("\nValeurs décimales sans tabulation: %d %d %d", a, b, c);
   printf("\nValeurs décimales avec tabulations: \t%d \t%d \t%d",
         a, b, c);
   printf("\nTrois types float sur 1 ligne: \t%f\t%f\t%f", f, g, h);
   printf("\nTrois types float sur 3 lignes: \n\t%f\n\t%f\n\t%f", f,
         g, h);
   printf("\nLe taux est de %f%%", f);
   printf("\nLe résultat de %f/%f est %f\n", g, f, g / f);
   exit(EXIT_FAILURE);
}
```



```
Valeurs décimales sans tabulation : 2 10 50
Valeurs décimales avec tabulations :   2   10   50
Trois types float sur une ligne :   1.050000   25.500000   -0.100000
Trois types float sur trois lignes :
    1.050000
    25.500000
    -0.100000
Le taux est de 1.050000%
Le résultat de 25.500000/1.050000 est 24.285715
```

Analyse

Les lignes 10 et 11 de ce programme affichent les trois décimales `a`, `b`, et `c`. La ligne 11 le fait en ajoutant des tabulations, absentes à la ligne 10. La ligne 13 du programme affiche les trois variables à virgule flottante `f`, `g` et `h` sur une ligne, la ligne 14 le fait sur trois lignes. La ligne 16 affiche la variable à virgule flottante `f` suivie du signe %. La ligne 17 illustre un dernier concept : une spécification de conversion peut être associée à une expression comme `g/f` ou même à une constante.



À ne pas faire

Coder plusieurs lignes de texte dans une seule instruction `printf()`. Le programme sera plus facile à lire si vous codez plusieurs instructions `printf()` contenant chacune une ligne de texte.

Omettre le caractère de retour à la ligne quand vous affichez plusieurs lignes avec des instructions `printf()` différentes.

Syntaxe de la fonction *printf()*

```
#include <stdio.h>
printf(chaîne-format [, arguments,...]);
```

La fonction `printf()` peut recevoir des arguments. Ceux-ci doivent correspondre, en nombre et en type, aux spécifications de conversion contenues dans la chaîne format. `printf()` envoie les informations mises en forme vers la sortie standard (l'écran). Pour qu'un programme puisse appeler la fonction `printf()`, le fichier standard d'entrées/sorties stdio.h doit être inclus.

La chaîne format peut contenir des ordres de contrôle. Le Tableau 7.1 donne la liste des ordres les plus souvent utilisés.

Voici quelques exemples d'appels de la fonction `printf()` :

Exemple 1 : code

```
#include <stdio.h>
int main()
{
    printf("Voici un exemple de message !");
    return 0;
}
```

Exemple 1 : résultat

```
Voici un exemple de message !
```

Exemple 2 : code

```
printf("Cela affiche un caractère, %c\nun nombre,
%d\nun nombre virgule \ flottante, %f", 'z', 123, 456.789);
```

Exemple 2 : résultat

```
Cela affiche un caractère, z
un nombre, 123
un nombre à virgule flottante, 456.789
```

La fonction *puts()*

La fonction `puts()` permet aussi d'afficher du texte à l'écran, mais pas de variable. Elle reçoit une chaîne de caractères en argument et l'envoie vers la sortie standard (écran), en ajoutant automatiquement un retour à la ligne à la fin du message. L'instruction :

```
puts("Hello, world.");
```

aura le même résultat que l'instruction :

```
printf("Hello, world.\n");
```

Vous pouvez inclure des ordres de contrôle dans une chaîne passée à la fonction `puts()`, ils ont la même signification que pour la fonction `printf()`.

Si votre programme utilise `puts()`, vous devez inclure le fichier en-tête stdio.h.



À faire

Utiliser la fonction `puts()` plutôt que `printf()` si le texte à afficher ne contient pas de variable.

À ne pas faire

Utiliser des spécifications de conversion dans une chaîne passée à la fonction `puts()`.

Syntaxe de la fonction *puts()*

```
#include <stdio.h>
puts (chaîne);
```

`puts()` est une fonction qui envoie une chaîne de caractères vers la sortie standard (écran). Pour l'utiliser, vous devez inclure le fichier en-tête stdio.h dans votre programme. La chaîne format de `puts()` peut contenir tous les ordres de contrôle de `printf()` (voir Tableau 7.1). Elle sera affichée à l'écran avec l'ajout d'un retour à la ligne à la fin du message.

Voici quelques exemples d'appels de cette fonction avec leurs résultats à l'écran :

Exemple 1 : code

```
puts("Cela est imprimé avec la fonction puts() !");
```

Exemple 1 : résultat

```
Cela est imprimé avec la fonction puts() !
```

Exemple 2 : code

```
puts("Première ligne du message. \nSeconde ligne du message.");
puts("Voici la troisième ligne.");
puts("Si nous avions utilisé printf(), ces 4 lignes auraient été \
sur 2 lignes !");
```

Exemple 2 : résultat

```
Première ligne du message.
Seconde ligne du message.
Voici la troisième ligne.
Si nous avions utilisé printf(), ces 4 lignes auraient été sur 2 lignes !
```

Lecture de données numériques avec *scanf()*

L'immense majorité des programmes a besoin d'afficher des données à l'écran et, de la même façon, de récupérer des données à partir du clavier. La façon la plus souple de le faire est d'utiliser la fonction `scanf()`.

La fonction `scanf()` lit les données entrées au clavier en fonction du format spécifié, et les attribue à une ou plusieurs variables du programme. Cette fonction utilise, comme `printf()`, une chaîne format qui décrit le format des données qui seront lues. La chaîne format contient les mêmes spécifications de conversion que pour la fonction `printf()`. Par exemple :

```
scanf("%d", &x);
```

Cette instruction lit un entier décimal et l'attribue à la variable entière `x`. De la même façon, l'instruction suivante lit une valeur avec virgule flottante et l'attribue à la variable `taux` :

```
scanf("%f", &taux);
```

Le symbole `&` est l'opérateur d'adresse du langage C. Ce concept est expliqué en détail dans le Chapitre 9. Retenez simplement que vous devez le placer devant le nom de chaque variable.

La chaîne reçue en argument par `scanf()` peut contenir un nombre illimité de variables. L'instruction suivante lit une variable entière et une autre à virgule flottante, et les attribue respectivement aux variables `x` et `taux` (sans oublier le signe `&`) :

```
scanf("%d %f", &x, &taux);
```

Quand la chaîne argument de `scanf()` contient plus d'une variable, un espace doit séparer les différents champs entrés au clavier. Cet espace peut être constitué d'un ou plusieurs blancs, de tabulations ou de lignes blanches. Cela vous donne une grande liberté sur la façon de saisir vos données. Chaque fois que `scanf()` lit un champ, elle le compare à la spécification de conversion correspondante.

En réponse à l'instruction `scanf()` précédente, vous auriez pu taper :

```
10  12.45
```

ou bien :

```
10   12.45
```

ou encore :

```
10
12.45
```

Comme pour les autres fonctions de ce chapitre, l'utilisation de `scanf()` implique celle du fichier en-tête stdio.h.

Listing 7.3 : Lecture de valeurs numérique avec scanf()

```
/* Exemple d'utilisation de la fonction scanf() */
#include <stdio.h>
#include <stdlib.h>

#define QUIT 4

int choix_menu(void);

int main()
 {
    int   choix    = 0;
    int   var_int   = 0;
    float var_float = 0.0;
    unsigned var_unsigned = 0;

    while(choix != QUIT)
    {
      choix = choix_menu();

```

Listing 7.3 : Lecture de valeurs numérique avec scanf() *(suite)*

```
      if(choix == 1)
      {
         puts("\nEntrez un entier décimal signé (ex -123)");
         scanf("%d", &var_int);
      }
      if (choix == 2)
      {
         puts("\nEntrez un nombre avec virgule flottante (ex 1.23)");
         scanf("%f", &var_float);
      }
      if (choix == 3)
      {
         puts("\nEntrez un entier décimal non signé (ex 123)");
         scanf("%u", &var_unsigned);
      }
    }
    printf("\nVos valeurs sont : int: %d  float: %f  unsigned: %u\n",
           var_int, var_float, var_unsigned);
    exit(EXIT_SUCCESS);
}

int choix_menu(void)
{
    int selection = 0;

    do
    {
      puts("\n1 - Lire un entier décimal signé");
      puts("2 - Lire un nombre avec virgule flottante");
      puts("3 - Lire un entier décimal non signé");
      puts("4 - Sortir");
      puts("\nEntrez votre choix :");

      scanf("%d", &selection);

      }while (selection < 1 || selection > 4);

    return selection;
}
```

```
1 - Lire un entier décimal signé
2 - Lire un nombre avec virgule flottante
3 - Lire un entier décimal non signé
4 - Sortir
Entrez votre choix :
1
Entrez un entier décimal signé (ex -123)
-123
1 - Lire un entier décimal signé
2 - Lire un nombre avec virgule flottante
```

```
3 - Lire un entier décimal non signé
4 - Sortir

Entrez votre choix :
3

Entrez un entier décimal non signé (ex 123)
321

1 - Lire un entier décimal signé
2 - Lire un nombre avec virgule flottante
3 - Lire un entier décimal non signé
4 - Sortir

Entrez votre choix :
2

Entrez un nombre avec virgule flottante (ex 1.23)
1231.123

1 - Lire un entier décimal signé
2 - Lire un nombre avec virgule flottante
3 - Lire un entier décimal non signé
4 - Sortir

Entrez votre choix :
4

Vos valeurs sont : int : -123 float : 1231.123047 unsigned : 321
```

Analyse

Le programme du Listing 7.3 utilise le même système de menu que celui du Listing 7.1. Les lignes 41 à 58 ont été quelque peu modifiées : la fonction `puts()` remplace la fonction `printf()` puisque le message ne contient pas de variable. Le caractère `\n` qui est ajouté automatiquement par `scanf()` a disparu des lignes 48 à 50. La ligne 55 a été transformée pour les 4 options de notre menu. La ligne 53 reste inchangée : `scanf()` lit une valeur décimale et la stocke dans la variable `selection`. La fonction renvoie alors `selection` au programme appelant en ligne 57.

Les programmes du Listing 7.1 et du Listing 7.3 ont la même structure de fonction `main()`. Une instruction `if` évalue le choix de l'utilisateur (valeur retournée par la fonction `choix menu()`) pour que le programme affiche le message correspondant. La fonction `scanf()` lit le nombre entré par l'utilisateur. Les lignes 23, 28 et 33 sont différentes parce que les variables lues ne sont pas du même type. Les lignes 12 à 14 contiennent les déclarations de ces variables.

Quand l'utilisateur décide de quitter le programme, celui-ci envoie un dernier message contenant la dernière valeur de chaque type qui a été lue. Si l'utilisateur n'a pas entré de valeur pour l'un d'entre eux, c'est la valeur d'initialisation 0 qui est affichée (lignes 12,

13, 14). Nous pouvons faire une dernière remarque concernant les lignes 20 à 34 : une structure de type `if...else` aurait été plus appropriée. Le Chapitre 14, qui approfondira le sujet de la communication avec l'écran, le clavier et l'imprimante, vous montrera que l'instruction idéale est une nouvelle instruction de contrôle : `switch`.



À faire

Utiliser conjointement `printf()` ou `puts()` avec `scanf()`. Les deux premières fonctions permettent de demander à l'utilisateur les variables que vous voulez lire.

À ne pas faire

Ne pas ajouter l'opérateur d'adresse (&) en codant les variables de `scanf()`.

Syntaxe de la fonction *scanf()*

```
#include <stdio.h>
scanf (chaîne format[ , arguments,...]);
```

La fonction `scanf()` lit des données entrées au clavier et utilise les spécifications de conversion de la chaîne format pour les attribuer aux différents arguments. Ces arguments représentent les adresses des variables plutôt que les variables elles-même. L'adresse d'une variable numérique est représentée par le nom de la variable précédé du signe (&). Un programme qui utilise `scanf()` doit faire appel au fichier en-tête stdio.h.

Exemple 1

```
int x, y, z;
scanf("%d %d %d", &x, &y, &z);
```

Exemple 2

```
#include <stdio.h>
int main()
{
    float y;
    int x;

    puts("Entrez un nombre entier puis un nombre à virgule flottante :");
    scanf("%f %d", &y, &x);
    printf("\nVous avez tapé %f et %d ", y, x);
    return 0;
}
```

Résumé

En combinant les trois fonctions `printf()`, `puts()` et `scanf()`, et les instructions de contrôle que nous avons étudiées aux chapitres précédents, vous avez tous les outils nécessaires pour écrire des programmes simples.

L'affichage d'informations sur l'écran se fait à partir de `puts()` et `printf()`. La fonction `puts()` affiche du texte exclusivement, la fonction `printf()` peut y ajouter des variables. Elles utilisent toutes deux des ordres de contrôle qui permettent d'afficher les caractères spéciaux et de mettre en forme le message envoyé.

La fonction `scanf()` lit au clavier une ou plusieurs valeurs numériques et les interprète en fonction de la spécification de conversion correspondante. Chaque valeur est attribuée à une variable du programme.

Q & R

Q Pourquoi utiliser `puts()` alors que `printf()` est moins restrictive ?

R La fonction `printf()` étant plus complète, elle consomme plus de ressources. Quand vos programmes vont se développer, les ressources vont devenir pré-cieuses. Il sera alors avantageux d'utiliser `puts()` pour du texte. En règle générale, utilisez plutôt la ressource disponible la plus simple.

Q Pourquoi faut-il inclure le fichier stdio.h quand on veut utiliser `printf()`, `puts()`, ou `scanf()` ?

R Le fichier stdio.h contient le prototype des fonctions standards d'entrée/sortie ; `printf()`, `puts()` et `scanf()` en font partie.

Q Que se passe-t-il si j'oublie l'opérateur d'adresse (`&`) d'une variable de la fonction `scanf()` ?

R Si vous oubliez cet opérateur, le résultat est totalement imprévisible. Au lieu de stocker la valeur reçue dans la variable, `scanf()` va la stocker à un autre endroit de la mémoire. Les conséquences peuvent être insignifiantes ou entraîner l'arrêt total de l'ordinateur. Vous comprendrez pourquoi après l'étude des Chapitres 9 et 13.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Quelle est la différence entre `puts()` et `printf()` ?
2. Quel fichier devez-vous inclure dans votre programme quand vous utilisez `printf()` ?
3. Que font les ordres de contrôle suivants :
   a) `\\`.
   b) `\b`.
   c) `\n`.
   d) `\t`.
   e) `\a`.
4. Quelle spécification de conversion faut-il utiliser si on veut afficher :
   a) Une chaîne de caractères.
   b) Un entier décimal signé.
   c) Un nombre décimal avec virgule flottante.
5. Quel est le résultat des séquences suivantes dans un texte passé à `puts()` ?
   a) `b`.
   b) `\b`.
   c) `\`.
   d) `\\`.

Exercices



À partir de ce chapitre, certains exercices vous demandent d'écrire un programme complet pour effectuer un certain travail. En langage C, il y a toujours plusieurs solutions à un problème donné : les réponses fournies en Annexe G ne sont pas les seules bonnes solutions. Si le code que vous avez développé ne génère pas d'erreur et donne le résultat recherché, vous avez trouvé une solution. Si vous rencontrez des difficultés, la solution donnée en exemple pourra vous aider.

1. Écrivez deux instructions qui provoquent un retour à la ligne, à l'aide des fonctions `printf()` et `puts()`.
2. Écrivez la fonction `scanf()` qui lira au clavier un caractère, un entier décimal non signé, puis un autre caractère.

3. Écrivez les instructions qui permettront de lire une valeur entière et de l’afficher à l’écran.
4. Transformez le code de l’exercice 3 pour qu’il n’accepte que des valeurs paires.
5. Modifiez l’exercice 4 pour qu’il renvoie des valeurs jusqu’à ce que le nombre 99 soit lu, ou que l’on ait lu la sixième valeur paire. Stockez les six nombres dans un tableau.
6. Transformez l’exercice 5 en code exécutable. Ajoutez une fonction qui affiche les valeurs du tableau, en les séparant par une tabulation, sur une seule ligne.
7. **CHERCHEZ L’ERREUR :**

```
printf("Jacques a dit, "Levez le bras droit !"");
```

8. **CHERCHEZ L’ERREUR :**

```
int lire_1_ou_2(void)
{
    int reponse = 0;
    while (reponse < 1 || reponse > 2)
    {
      printf(Entrez 1 pour oui, 2 pour non);
      scanf("%f", reponse);
    }
    return reponse;
}
```

9. Complétez le Listing 7.1 pour que la fonction affiche le Tableau 7.1 en entier.
10. Écrivez un programme qui lira deux nombres avec virgule flottante au clavier, puis affichera leur produit.
11. Écrivez un programme qui lira dix valeurs entières au clavier et affichera leur somme.
12. Écrivez un programme qui lira des entiers au clavier pour les stocker dans un tableau. L’entrée des données s’arrêtera si l’utilisateur tape 0 ou si le tableau est complet. Le programme affichera ensuite la plus petite et la plus grande valeur du tableau.



Ce problème est difficile, car nous n’avons pas fini d’étudier les tableaux. Si vous ne trouvez pas de solution, essayez de faire cet exercice après avoir lu le Chapitre 8.

Révision de la Partie I

Cette première partie vous a familiarisé avec la programmation : coder, compiler, effectuer la liaison et exécuter un programme. Le programme qui suit reprend de nombreux sujets déjà étudiés.

La philosophie de cette section diffère de celle des exemples pratiques. Vous ne trouverez pas d'éléments inconnus dans le programme. Et celui-ci est suivi d'une analyse. Les Parties II et III sont aussi suivies de sections de ce type.



Les indications dans la marge vous renvoient au chapitre qui présente les concepts de la ligne. Vous pourrez ainsi retrouver les informations correspondantes.

Listing révision de la Partie I

Ch. 02

```
 /* Nom du programme : week1.c                          */
 /*   Ce programme permet de saisir l'âge et le revenu  */
 /*   de 100 personnes au maximum. Il affiche ensuite   */
 /*   les résultats obtenus                             */
 /*----------------------------------------------------*/
 /*---------------------------*/
 /* Fichiers inclus           */
 /*---------------------------*/
```

Ch. 02

```
#include <stdio.h>
#include <stdlib.h>
```

Ch. 02

```
/*---------------------------*/
/* Définition des constantes */
/*---------------------------*/
```

```
        
Ch. 02  #define MAX    100
        #define OUI    1
        #define NON    0
        
Ch. 02  /*---------------------------*/
        /*   Variables               */
        /*---------------------------*/
        
Ch. 03  long revenu[MAX];              /* pour stocker les revenus */
        int  mois[MAX], jour[MAX], annee[MAX];
                                        /* pour les dates de naissance */
        int  x, y, ctr;                /* compteurs */
        int  cont;                     /* contrôle du programme */
        long total_mois, grand_total;  /* calcul des totaux */
        
Ch. 02  /*---------------------------*/
        /* Prototypes des fonctions  */
        /*---------------------------*/
        
Ch. 05  int main(void);
        int affiche_instructions(void);
        void lecture(void);
        void affiche_result(void);
        int continuer(void);
        
        /*---------------------------*/
        /*   Début du programme      */
        /*---------------------------*/
        int main(void)
Ch. 02  {
            cont = affiche_instructions();
Ch. 05  
            if (cont == OUI)
Ch. 04      {
              lecture();
Ch. 05        affiche_result();
Ch. 05      }
            else
Ch. 04        printf("\nProgramme interrompu par l'utilisateur !\n\n");
Ch. 07      exit(EXIT_SUCCESS);
        }
Ch. 02  /*-----------------------------------------------------------*/
        /* Fonction : affiche_instructions()                         */
        /* objectif : affiche le mode d'emploi du programme et       */
        /*            demande à l'utilisateur d'entrer 0 pour        */
        /*            sortir ou 1 pour continuer                     */
        /* Valeurs renvoyées : NON  si l'utilisateur tape 0          */
        /*                     OUI  si l'utilisateur tape un nombre  */
        /*                          différent de 0                   */
        /*-----------------------------------------------------------*/
Ch. 05  
        int affiche_instructions(void)
```

```
        {
Ch. 07      printf("\n\n");
            printf("\nCe programme vous permet de saisir le revenu et");
            printf("\nla date de naissance de 99 personnes maxi, pour");
            printf("\ncalculer et afficher le total des revenus mois par mois,");
            printf("\nle total annuel des revenus, et la moyenne de ces revenus.");
            printf("\n");

Ch. 05      cont = continuer();

Ch. 05      return(cont);
        }
Ch. 02  /*-----------------------------------------------------------*/
        /* Fonction : lecture()                                      */
        /* Objectif : Cette fonction lit les données entrées par     */
        /*    l'utilisateur jusqu'à ce que 100 personnes soient      */
        /*    enregistrées, ou que l'utilisateur tape 0 pour le mois */
        /* Valeurs renvoyées : aucune                                */
        /* Remarque : Cela permet d'entrer 0/0/0 dans le champ       */
        /*    anniversaire si l'utilisateur ne le connaît pas.       */
        /*    Cela autorise également 31 jours pour tous les mois    */
        /*-----------------------------------------------------------*/

Ch. 05  void lecture(void)
        {
Ch. 06      for (cont = OUI, ctr = 0; ctr < MAX && cont == OUI; ctr++)
            {
Ch. 07        printf("\nEntrez les informations pour la personne no %d", ctr+1);
              printf("\n\tDate de naissance :");

Ch. 06        do
              {
Ch. 07        printf("\n\tMois (0 - 12): ");
Ch. 07       scanf("%d", &mois[ctr]);
Ch. 06         }while (mois[ctr] < 0 || mois[ctr] > 12);

Ch. 06        do
              {
Ch. 07          printf("\n\tJour (0 - 31): ");
Ch. 07          scanf("%d", &jour[ctr]);
Ch. 06        }while (jour[ctr] < 0 || jour[ctr] >31);

Ch. 06        do
              {
Ch. 07          printf("\n\tAnnée (0 - 1994): ");
Ch. 07          scanf("%d", &annee[ctr]);
Ch. 06        }while (annee[ctr] < 0 || annee[ctr] > 1994);

Ch. 07        printf("\nEntrez le revenu annuel (en francs): ");
              scanf("%ld%", &revenu[ctr]);

Ch. 05        cont = continuer();
            }
```

```
Ch. 07  /* La valeur de ctr correspond au nombre de personnes enregistrées*/
        }
Ch. 02  /*------------------------------------------------------------*/
        /* Fonction : affiche_result()                                */
        /* Objectif : affiche le résultat des calculs à l'écran       */
        /* Valeurs renvoyées : aucune                                 */
        /*------------------------------------------------------------*/

Ch. 05  void affiche_result()
        {
Ch. 04      grand_total = 0;
Ch. 07      printf("\n\n\n");    /* on saute quelques lignes */
            printf("\n    Salaires");
            printf("\n    ========");

Ch. 06      for (x = 0; x <= 12; x++) /* pour chaque mois */
            {
                total_mois = 0;
Ch. 04          for (y = 0; y < ctr; y++)
Ch. 06          {
                    if(mois[y] == x)
Ch. 04                total_mois += revenu[y];
Ch. 04          }
                printf("\nLe total pour le mois %d est %ld", x, total_mois);
Ch. 07          grand_total += total_mois;
Ch. 04      }
            printf("\n\n\nLe total des revenus est de %ld", grand_total);
Ch. 07      printf("\nLa moyenne des revenus est de %ld", grand_total/ctr);

            printf("\n\n* * * fin des résultats * * *");
        }
Ch. 02  /*-------------------------------------------------------------*/
        /* Fonction : continuer()                                      */
        /* Objectif : cette fonction demande à l'utilisateur s'il      */
        /*            veut continuer                                   */
        /* Valeurs renvoyées : OUI si l'utilisateur désire poursuivre*/
        /*                     NON si l'utilisateur veut sortir        */
        /*-------------------------------------------------------------*/
Ch. 05  int continuer(void)
        {
Ch. 07      printf("\n\nVoulez-vous continuer ? (0=non / 1=oui) :");
            scanf("%d", &x);

Ch. 06      while (x < 0 || x > 1)
            {
Ch. 07          printf("\n%d est erroné !", x);
                printf("\nEntrez 0 pour sortir ou 1 pour continuer :");
                scanf("%d", &x);
            }
            if (x == 0)
                return(NON);
            else
                return(OUI);
        }
```

Analyse

Quand vous aurez répondu aux quiz et exercices des Chapitres 1 et 2, vous saurez saisir et compiler le code de ce programme. Nous nous sommes placés en mode réel de travail en commentant abondamment les différentes parties, et en particulier, le début du programme et chaque fonction majeure. Les lignes 1 à 5 contiennent le nom et le descriptif du programme. Certains programmeurs ajouteraient des informations comme le nom de l'auteur du programme, le compilateur utilisé avec son numéro de version, les bibliothèques liées au programme et sa date de création. Les commentaires précédant chaque fonction indiquent la tâche réalisée par cette fonction, éventuellement le type de valeur renvoyée ou les conventions d'appel.

Les commentaires du début vous indiquent que vous pouvez saisir des renseignements concernant 100 personnes au maximum. Avant de commencer la lecture des données tapées par l'utilisateur, le programme appelle `affiche instructions()` (ligne 44). Cette fonction affiche le descriptif du programme et demande à l'utilisateur s'il est d'accord pour continuer. Le code de cette fonction utilise `printf()` (lignes 67 à 72) que vous avez étudiée au Chapitre 7.

La fonction `continuer()` en lignes 157 à 172 utilise quelques notions étudiées en fin de partie. La ligne 159 demande à l'utilisateur s'il veut continuer. L'instruction de contrôle `while` vérifie la réponse et renvoie le message jusqu'à obtention d'une réponse valide : 0 ou 1. L'instruction `if   else` renvoie alors au programme la variable `OUI` ou `NON`.

Le principal travail réalisé par le programme dépend de deux fonctions : `lecture()` et `affiche()`. La première vous demande d'entrer les données pour les stocker dans les tableaux déclarés en début de programme. L'instruction `for` de la ligne 91 lit les données jusqu'à ce que `cont` soit différent de la constante `OUI` (renvoyée par la fonction `continue()`) ou que la valeur du compteur `ctr` soit supérieure ou égale à la valeur `MAX` (nombre maximum d'éléments dans les tableaux). Le programme contrôle et valide chaque information saisie. Les lignes 96 à 100, par exemple, demandent à l'utilisateur d'entrer un numéro de mois. Si la valeur saisie n'est pas un entier compris entre 0 et 12 inclus, le message est affiché de nouveau. La ligne 117 appelle la fonction `continuer()` pour savoir si l'utilisateur désire continuer ou arrêter.

Si la réponse est 0, ou si le nombre maximum d'enregistrements est atteint (MAX), l'exécution se poursuit en ligne 49 avec l'appel de la fonction `affiche result()`. La fonction `affiche result()` des lignes 127 à 149 envoie un compte rendu des enregistrements à l'écran. Une boucle `for` imbriquée permet le calcul du total des revenus mois par mois et du total général.

Ce programme illustre les sujets étudiés au cours de cette première partie. Vos connaissances concernant le langage C sont encore limitées, mais vous êtes maintenant capable d'écrire vos propres programmes.

Tour d'horizon de la Partie II

Votre première partie d'étude de la programmation en langage C est terminée. Vous êtes maintenant familiarisé avec l'éditeur pour saisir vos programmes, et le compilateur pour créer le code exécutable correspondant.

Ce que vous allez apprendre

Cette seconde partie d'apprentissage couvre de nombreux concepts qui constituent le cœur du langage C. Vous allez apprendre à utiliser les tableaux numériques et les tableaux de caractères, et à créer des structures pour grouper différents types de variables.

Cette deuxième partie introduit de nouvelles instructions de contrôle, et fournit un descriptif détaillé de diverses fonctions.

Les Chapitres 9 et 12 traitent de sujets très importants pour comprendre les principes du développement en langage C : les pointeurs et la portée des variables.

Après les programmes simples de la première partie, les informations fournies par la deuxième vous permettront d'écrire des programmes plus complexes pour accomplir presque toutes les tâches.

8

Utilisation des tableaux numériques

Les tableaux représentent un type de stockage de données souvent utilisé en langage C. Le Chapitre 6 vous en a donné un bref aperçu. Aujourd'hui, vous allez étudier :

- La définition d'un tableau
- Les tableaux numériques à une ou plusieurs dimensions
- La déclaration et l'initialisation des tableaux

Définition

Un tableau représente un ensemble d'emplacements mémoire qui portent le même nom et contiennent le même type de données. Chacun de ces emplacements est un *élément du tableau*. Pour démontrer l'utilité des tableaux, le mieux est de prendre un exemple. Si vous gardez une trace de vos frais professionnels pour 1998 en classant vos reçus mois par mois, vous pourriez constituer un dossier par mois. Toutefois, il serait sûrement plus pratique d'avoir un seul dossier comportant douze compartiments.

Adaptons cet exemple à la programmation. Supposons que vous écriviez un programme pour le calcul de vos frais professionnels. Ce programme pourrait déclarer douze variables différentes, correspondant aux douze mois de l'année. Un bon programmeur utilisera plutôt un tableau de douze éléments, où le total de chaque mois est stocké dans l'élément correspondant.

La Figure 8.1 vous montre la différence entre l'utilisation de variables individuelles et un tableau.

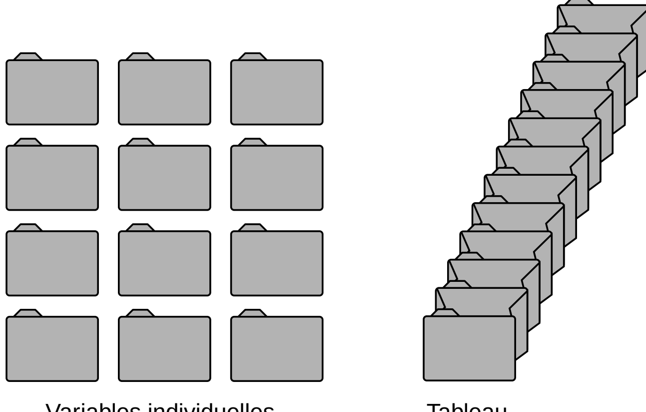


Figure 8.1
Les variables sont l'équivalent de dossiers indi-viduels, alors que le tableau représente un dossier ayant de multiples compartiments.

Les tableaux à une dimension

Un *tableau à une dimension* ne possède qu'un seul index. Un *index* est le nombre entre crochets qui suit le nom du tableau. Il indique le nombre d'éléments du tableau. Dans le cas du programme de calcul de vos frais professionnels, par exemple, vous pourriez déclarer un tableau de type `float` :

```
float depenses[12];
```

Le tableau s'appelle `depenses` et contient 12 éléments, chacun d'eux étant l'équivalent d'une variable de type `float`. Un élément de tableau peut contenir n'importe quel type de

donnée du langage C. Les éléments sont toujours numérotés en commençant à 0 ; ceux de notre exemple seront donc numérotés de 0 à 11.

Pour chaque déclaration de tableau, le compilateur réserve un bloc de mémoire d'une taille suffisante pour contenir la totalité des éléments. Ceux-ci seront stockés séquentiellement comme le montre la Figure 8.2.

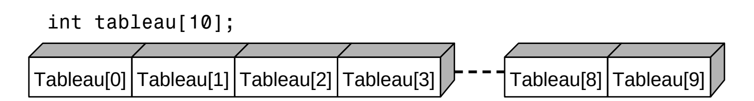


Figure 8.2
Les éléments de tableau sont stockés en mémoire de façon séquentielle.

Comme pour les variables simples, l'emplacement de la déclaration du tableau dans le code source est important. Il détermine la façon dont le programme pourra utiliser le tableau. En attendant que le Chapitre 12 vous donne les informations nécessaires, positionnez vos déclarations avec les autres déclarations de variables, avant le début de la fonction principale `main()`.

Un élément de tableau s'utilise comme une variable isolée de même type. Il est référencé au moyen du nom de tableau suivi de son index entre crochets. L'instruction suivante, par exemple, stocke la valeur 89,95 dans le second élément de notre tableau `depenses` :

```
depenses[1] = 89.95;
```

De la même façon, l'instruction :

```
depenses[10] = depenses[11];
```

stocke un exemplaire de la valeur contenue dans l'élément de tableau `depenses[11]` dans l'élément de tableau `depenses[10]`. L'index du tableau peut être une constante comme dans notre exemple, mais aussi une expression, une variable entière, ou même un autre élément de tableau. Voici quelques exemples :

```
float depenses[100];
int a[10];
/* Des instructions peuvent prendre place ici */
depenses[i]=100;       /* i est une variable entière */
depenses[2 + 3] = 100; /* équivalent de depenses[5] */
depenses[a[2]] = 100;  /* a[] est un tableau contenant des entiers */
```

La dernière ligne nécessite un complément d’information. Si `a[]` est un tableau contenant des entiers, et que la valeur 8 est stockée dans l’élément `a[2]`, écrire :

```
depenses[ a[2] ]
```

a la même signification que :

```
depenses[8]
```

N’oubliez pas que, dans un tableau de *n* éléments, l’index est compris entre 0 et *n*–1. L’emploi de la valeur d’index *n* ne sera pas décelé par le compilateur, mais cela provoquera des erreurs dans les résultats du programme.



Les éléments d’un tableau commencent à 0 et non à 1. Un tableau contenant 10 éléments, par exemple, commencera à 0 et se terminera à 9.

Vous pouvez, pour vous simplifier la tâche, adresser les éléments d’un tableau en commençant à 1 jusqu’à n. La méthode la plus simple consiste à déclarer un tableau avec n + 1 éléments et d’en ignorer le premier.

Listing 8.1 : depenses.c illustre l’utilisation d’un tableau

```
/* depenses.c -- Exemple d'utilisation d'un tableau */
#include <stdio.h>
#include <stdlib.h>

/*Déclaration du tableau pour enregistrer les dépenses, */
/* et d'une variable compteur */
float depenses[13];
int compteur;

int main()
{
/* Lecture des données au clavier et stockage dans le tableau */

   for (compteur = 1; compteur < 13; compteur++)
   {
     printf("Entrez les dépenses pour le mois %d : ", compteur);
     scanf("%f", &depenses[compteur]);
   }

/* Affichage du contenu du tableau */

   for (compteur = 1; compteur < 13; compteur++)
   {
      printf("Mois %d = %.2fF\n", compteur, depenses[compteur]);
   }
   exit(EXIT_SUCCESS);
}
```

```
Entrez les dépenses pour le mois  1 : 100
Entrez les dépenses pour le mois  2 : 200.12
Entrez les dépenses pour le mois  3 : 150.50
Entrez les dépenses pour le mois  4 : 300
Entrez les dépenses pour le mois  5 : 100.50
Entrez les dépenses pour le mois  6 : 34.25
Entrez les dépenses pour le mois  7 : 45.75
Entrez les dépenses pour le mois  8 : 195.00
Entrez les dépenses pour le mois  9 : 123.45
Entrez les dépenses pour le mois 10 : 111.11
Entrez les dépenses pour le mois 11 : 222.20
Entrez les dépenses pour le mois 12 : 120.00

Mois  1 = 100.00F
Mois  2 = 200.12F
Mois  3 = 150.50F
Mois  4 = 300.00F
Mois  5 = 100.50F
Mois  6 = 34.25F
Mois  7 = 45.75F
Mois  8 = 195.00F
Mois  9 = 123.45F
Mois 10 = 111.11F
Mois 11 = 222.20F
Mois 12 = 120.00F
```

Analyse

Quand vous exécutez ce programme, un message vous demande d'entrer les dépenses correspondant aux douze mois de l'année ; les valeurs saisies sont alors affichées à l'écran.

La ligne 1 contient un descriptif du programme. Inclure le nom du programme dans les commentaires d'en-tête pourra vous être très utile si par exemple vous imprimez le programme pour le modifier.

La ligne 5 annonce la déclaration des différentes variables, et la ligne 7 contient la déclaration d'un tableau de 13 éléments. Ce programme n'a besoin que de 12 éléments (pour les 12 mois de l'année), mais nous en avons déclaré 13. La boucle `for` des lignes 14 à 18 ignore l'élément 0. La variable `compteur` déclarée en ligne 8 sera utilisée comme compteur et comme index pour le tableau.

La fonction principale `main()` commence en ligne 10. Une boucle `for` demande à l'utilisateur la valeur des dépenses pour chacun des 12 mois. La fonction `scanf()` de la ligne 17 range cette valeur dans un élément du tableau. `%f` est utilisé parce que le tableau `depenses` a été déclaré avec le type `float` en ligne 7. L'opérateur d'adresse `(&)` est placé devant l'élément de tableau, exactement comme si c'était une variable `float`.

Les lignes 22 à 25 contiennent une seconde boucle `for` qui affiche les valeurs du tableau. `%.2f` permet d'afficher un nombre avec deux chiffres après la virgule. Le Chapitre 14 traite de ce type de commande qui permet de mettre en forme le texte à afficher.



À faire

Utiliser un tableau plutôt que créer plusieurs variables pour stocker le même type de données.

À ne pas faire

N'oubliez pas que l'index d'un tableau commence à la valeur 0.

Les tableaux à plusieurs dimensions

Un tableau à plusieurs dimensions possède plusieurs index. Un tableau à deux dimensions en a deux, un tableau à trois dimensions en a trois, etc. En langage C, il n'y a pas de limite au nombre de dimensions qu'un tableau peut avoir.

Vous pouvez, par exemple, écrire un programme qui joue aux échecs. L'échiquier contient 64 carrés sur huit lignes et huit colonnes. Votre programme pourra le représenter sous forme de tableau à deux dimensions de la façon suivante :

```
int échiquier[8][8];
```

Le tableau ainsi créé a 64 éléments : `échiquier[0][0]`, `échiquier[0][1]`, `échiquier[0][2]`… `échiquier[7][6]`, `échiquier[7][7]`. La Figure 8.3 représente la structure de ce tableau.

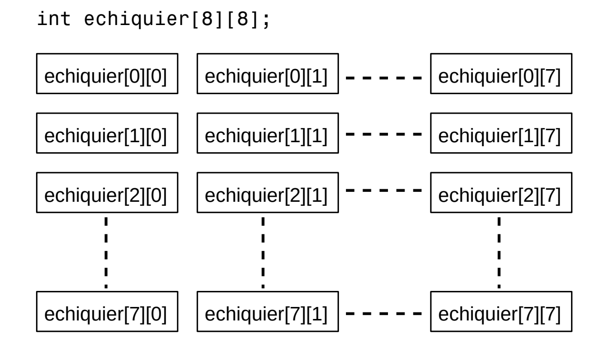


Figure 8.3
Un tableau à deux dimensions a une structure en lignes-colonnes.

On peut aussi imaginer un tableau à trois dimensions comme un cube. Quel que soit le nombre de ses dimensions, un tableau est stocké en mémoire de façon séquentielle.

Le nom et la déclaration des tableaux

Les règles concernant l'attribution d'un nom à un tableau sont les mêmes que celles qui régissent les noms de variables (voir Chapitre 3). Un nom de tableau doit être unique : il ne doit pas avoir été attribué précédemment à un autre tableau, une variable ou une constante, etc. Une déclaration de tableau a la même forme qu'une déclaration de variable, mis à part le fait que le nombre d'éléments du tableau doit apparaître entre crochets immédiatement après son nom.

Dans la déclaration, le nombre d'éléments du tableau peut être une constante littérale ou une constante symbolique créée avec l'ordre `#define`. Exemple :

```
#define MOIS 12
int tableau[MOIS];
```

L'instruction qui suit est équivalente :

```
int tableau[12];

const int MOIS = 12;
```

Listing 8.2 : Le programme notes.c stocke dix notes dans un tableau

```
/* notes.c--Echantillon d'un programme qui utilise un tableau */
/* pour lire 10 notes et en calculer la moyenne */
#include <stdio.h>
#include <stdlib.h>

#define MAX_NOTE 100
#define ETUDIANTS  10

int notes[ETUDIANTS];

int idx;
int total = 0;           /* pour le calcul de la moyenne */

int main()
{
   for(idx = 0; idx < ETUDIANTS; idx++)
   {
      printf("Entrez la note de l\'étudiant numero %d : ", idx+1);
      scanf("%d", &notes[idx]);

      while (notes[idx] > MAX_NOTE)
      {
         printf("\nLa note maximum est %d", MAX_NOTE);
         printf("\nEntrez une note correcte : ");
         scanf("%d", &notes[idx]);
      }
```

Listing 8.2 : Le programme notes.c stocke dix notes dans un tableau *(suite)*

```
27:
28:         total += notes[idx];
29:     }
30:
31:     printf("\n\nLa moyenne des notes est %d\n",
32:              (total / ETUDIANTS));
33:     exit(EXIT_SUCCESS);
34: }
```

```
Entrez la note de l'étudiant numéro  1 : 95
Entrez la note de l'étudiant numéro  2 : 100
Entrez la note de l'étudiant numéro  3 : 60
Entrez la note de l'étudiant numéro  4 : 105

La note maximum est 100
Entrez une note correcte : 100
Entrez la note de l'étudiant numéro  5 : 25
Entrez la note de l'étudiant numéro  6 : 0
Entrez la note de l'étudiant numéro  7 : 85
Entrez la note de l'étudiant numéro  8 : 85
Entrez la note de l'étudiant numéro  9 : 95
Entrez la note de l'étudiant numéro 10 : 85

La moyenne des notes est 73
```

Analyse

Ce programme demande à l'utilisateur d'entrer les notes de dix étudiants, puis il en affiche la moyenne.

Le tableau du programme s'appelle `notes` (ligne 9). Deux constantes sont définies aux lignes 6 et 7 : `MAX NOTE` et `ETUDIANTS`. La valeur de ces constantes pourra changer facilement. Celle de la constante `ETUDIANTS` étant 10, cela représente aussi le nombre d'éléments du tableau. La variable `idx`, abréviation d'index, est utilisée comme compteur et comme index du tableau. La variable `total` contiendra la somme de toutes les notes.

Le travail principal du programme se fait aux lignes 16 à 29 avec la boucle `for`. La variable `idx` est initialisée à 0, le premier indice du tableau ; elle est incrémentée à chaque exécution de la boucle qui lit la note d'un étudiant (lignes 18 et 19). Remarquez que la ligne 18 ajoute 1 à la valeur d'`idx` de façon à compter de 1 à 10 plutôt que de 0 à 9 : la première note est stockée en `notes[0]`, mais on a demandé à l'utilisateur la note de l'étudiant numéro 1.

Les lignes 21 à 26 contiennent une boucle `while` imbriquée dans la boucle `for`. Cette boucle permet de vérifier la validité de la note donnée par l'utilisateur. Si elle est supérieure à MAX_NOTE, un message demande à l'utilisateur de retaper une note correcte.

La ligne 28 additionne les notes à chaque exécution de la boucle, et la ligne 31 affiche la moyenne de ces notes en fin d'exécution du programme.



À faire

Utiliser des instructions `#define` *pour créer les constantes qui permettront de déclarer les tableaux. Il sera ainsi facile de changer la valeur du nombre d'éléments. Dans le programme notes.c, par exemple, vous pouviez changer le nombre d'étudiants dans l'instruction* `#define` *sans avoir à changer le reste du programme.*

À ne pas faire

Créer des tableaux avec plus de trois dimensions. Ils peuvent rapidement devenir trop importants.

Initialisation

Vous pouvez initialiser l'intégralité ou seulement une partie d'un tableau, au moment de sa déclaration. Il faut prolonger la déclaration du tableau d'un signe égal suivi d'une liste entre accolades de valeurs séparées par des virgules. Ces valeurs sont attribuées dans l'ordre aux éléments du tableau en commençant à l'élément 0. Par exemple, l'instruction suivante attribue la valeur 100 à `tableau[0]`, 200 à `tableau[1]`, 300 à `tableau[2]` et 400 à `tableau[3]` :

```
int tableau[4] = { 100, 200, 300, 400 };
```

Si vous n'indiquez pas la taille du tableau, le compilateur va créer un tableau avec autant d'éléments que de valeurs initiales. L'instruction suivante est donc parfaitement équivalente à la précédente :

```
int tableau[] = { 100, 200, 300, 400 };
```

Si les éléments d'un tableau ne sont pas initialisés en début de programme, vous ne connaissez pas les valeurs qui y sont stockées quand le programme s'exécute. Si vous initialisez plus d'éléments que le tableau n'en contient, le compilateur envoie un message d'erreur.

Initialisation de tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions peuvent aussi être initialisés. Les valeurs sont attribuées séquentiellement en incrémentant d'abord le dernier index :

```
int tableau[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Cette instruction affecte les valeurs dans l'ordre suivant :

```
tableau[0][0] = 1
tableau[0][1] = 2
tableau[0][2] = 3
tableau[1][0] = 4
tableau[1][1] = 5
tableau[1][2] = 6
etc.
tableau[3][1] = 11
tableau[3][2] = 12
```

Quand vous initialisez un tableau à plusieurs dimensions, vous pouvez rendre le code source plus clair en groupant les valeurs entre des accolades supplémentaires, et en les répartissant sur plusieurs lignes. Notre exemple précédent pourrait être réécrit de cette façon :

```
int tableau[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,
{ 7, 8, 9 } , { 10, 11, 12 } };
```

Il ne faut pas oublier la virgule qui doit séparer les valeurs, même si elles sont déjà séparées par des accolades.

Le Listing 8.3 crée un tableau à trois dimensions de 1000 éléments et y stocke des nombres de manière aléatoire. Le programme affiche ensuite le contenu des éléments du tableau. C'est un bon exemple de l'intérêt qu'offre un tableau. Imaginez le nombre de lignes de code qui auraient été nécessaires pour effectuer la même tâche avec des variables.

Ce programme contient une nouvelle fonction de bibliothèque : `getch()`. Cette fonction lit un caractère au clavier. Dans notre exemple, elle met le programme en pause jusqu'à ce que l'utilisateur enfonce une touche du clavier. Cette fonction est décrite en détail au Chapitre 14.

Listing 8.3 : Le programme alea.c crée un tableau à plusieurs dimensions

```
/* alea.c -- Exemple d'utilisation d'un tableau à plusieurs  */
/* dimensions                                                 */
#include <stdio.h>
#include <stdlib.h>
/* Déclaration d'un tableau à 3 dimensions de 1000 éléments  */

int random[10][10][10];
int a, b, c;

int main()
{
/* On remplit le tableau avec des nombres aléatoires.       */
/* La fonction de bibliothèque rand() renvoi un nombre      */
```

```
/* aléatoire. On utilise une boucle for pour chaque indice. */

   for (a = 0; a < 10; a++)
   {
      for (b = 0; b < 10; b++)
      {
         for (c = 0; c < 10; c++)
         {
            random[a][b][c] = rand();
         }
      }
   }

/* On affiche les éléments du Tableau 10 par 10 */

   for (a = 0; a < 10; a++)
   {
      for (b = 0; b < 10; b++)
      {
         for (c = 0; c < 10; c++)
         {
            printf("\nrandom[%d][%d][%d] = ", a, b, c);
            printf("%d", random[a][b][c]);
         }
         printf("\nAppuyez sur Entrée pour continuer, CTRL-C pour sortir.");

         getchar();
      }
   }
   exit(EXIT_SUCCESS);
}  /* fin de la fonction main() */
```

```
random[0][0][0] = 346
random[0][0][1] = 130
random[0][0][2] = 10982
random[0][0][3] = 1090
random[0][0][4] = 11656
random[0][0][5] = 7117
random[0][0][6] = 17595
random[0][0][7] = 6415
random[0][0][8] = 22948
random[0][0][9] = 31126
Appuyez sur Entrée pour continuer,
ou CTRL-C pour sortir.
random[0][1][0] = 346
random[0][1][1] = 130
random[0][1][2] = 10982
random[0][1][3] = 1090
random[0][1][4] = 11656
random[0][1][5] = 7117
random[0][1][6] = 17595
random[0][1][7] = 6415
random[0][1][8] = 22948
```

```
random[0][1][9] = 31126
Appuyez sur Entrée pour continuer,
ou CTRL-C pour sortir.
etc.
random[9][8][0] = 6287
random[9][8][1] = 26957
random[9][8][2] = 1530
random[9][8][3] = 14171
random[9][8][4] = 6957
random[9][8][5] = 213
random[9][8][6] = 14003
random[9][8][7] = 29736
random[9][8][8] = 15028
random[9][8][9] = 18968
Appuyez sur Entrée pour continuer,
ou CTRL-C pour sortir.
random[9][9][0] = 28559
random[9][9][1] = 5268
random[9][9][2] = 10182
random[9][9][3] = 3633
random[9][9][4] = 24779
random[9][9][5] = 3024
random[9][9][6] = 10853
random[9][9][7] = 28205
random[9][9][8] = 8930
random[9][9][9] = 2873
Appuyez sur Entrée pour continuer,
ou CTRL-C pour sortir.
```

Analyse

Au Chapitre 6, nous avons étudié un programme qui utilisait une boucle `for` imbriquée. Celui-ci en a deux. Les lignes 7 et 8 contiennent les définitions de 4 variables : `random` est un tableau à trois dimensions qui stockera des nombres entiers aléatoires et qui contient 1000 éléments ($10 \times 10 \times 10$). La ligne 8 déclare les 3 variables `a`, `b`, et `c` destinées au contrôle des boucles.

En ligne 4, ce programme inclut un fichier en-tête dont nous n'avons que peu parlé jusqu'ici, stdlib.hstdlib.h (STanDart LIBrary). Il contient le prototype de la fonction `rand()` utilisée à la ligne 22. C'est également lui qui définit la constante EXIT_SUCCESS (ligne 43) et son pendant EXIT_FAILURE.

Les deux instructions `for` imbriquées représentent la partie principale du programme. La première, aux lignes 16 à 25, a la même structure que la deuxième aux lignes 29 à 42. La ligne 22 de la première boucle s'exécute de façon répétitive, et alimente le tableau `random` avec les nombres aléatoires fournis par la fonction `rand()`.

Si nous remontons un peu dans le listing, nous pouvons noter que la boucle `for` de la ligne 20 va s'exécuter 10 fois pour des valeurs de la variable `c` allant de 0 à 9. Cette boucle représente l'index le plus à droite du tableau `random`. La ligne 18 est la boucle de `b`, qui

représente l’index du milieu du tableau. Enfin la ligne 16 incrémente la variable `a` pour l’index de gauche du tableau `random`. À chaque changement de la valeur de `a`, la boucle contenant la variable `b` s’exécute 10 fois et à chaque exécution de cette boucle, celle qui incrémente `c` a tourné aussi 10 fois. Ces boucles permettent donc d’initialiser tous les éléments de `random`.

Les lignes 29 à 42 contiennent la seconde série de boucles `for` imbriquées. Le principe est exactement le même que pour les trois boucles précédentes. Cette fois, leur tâche consiste à afficher, par groupe de 10, les valeurs précédemment initialisées. À la fin de chaque série, les lignes 38 et 39 demandent à l’utilisateur s’il veut continuer. La fonction `getchar()` suspend le programme jusqu’à ce que l’on appuie sur Entrée. Lancez ce programme et regardez les valeurs affichées.

Taille maximale

La mémoire occupée par un tableau dépend du nombre et de la taille des éléments qu’il contient. La taille d’un élément dépend de la taille, sur votre ordinateur, du type de donnée qu’il contient. Le Tableau 8.1 reprend les tailles qui avaient été attribuées au Chapitre 3.

Tableau 8.1 : Espace mémoire nécessaire pour stocker les données numériques

Type de la donnée stockée dans l’élément	*Taille de l’élément (en octets)*
`int`	4
`short`	2
`long`	4
`float`	4
`double`	8

L’espace mémoire peut être calculé à l’intérieur d’un programme au moyen de l’opérateur `sizeof()`.C’est un opérateur unaire, et non une fonction. Il prend le nom d’une variable ou d’un type de donnée comme argument et en renvoie la taille en octets.

Listing 8.4 : Utilisation de l’opérateur sizeof() pour calculer l’espace occupé par un tableau

```
/* Exemple d'utilisation de l'opérateur sizeof() */
#include <stdio.h>
#include <stdlib.h>

/* On déclare quelques tableaux de 100 éléments */

```

Listing 8.4 : Utilisation de l'opérateur sizeof() pour calculer l'espace occupé par un tableau *(suite)*

```
 7:  int inttab[100];
 8:  float floattab[100];
 9:  double doubletab[100];
10:
11:  int main()
12:  {
13:  /* On affiche la taille des types de données  */
14:
15:     printf("\n\nLa taille de int est de %d octets", sizeof(int));
16:     printf("\nLa taille de short est de %d octets", sizeof(short));
17:     printf("\nLa taille de long est de %d octets", sizeof(long));
18:     printf("\nLa taille de float est de %d octets", sizeof(float));
19:     printf("\nLa taille de double est de %d bytes", sizeof(double));
20:
21:  /* On affiche la taille des trois tableaux */
22:
23:     printf("\nTaille de inttab = %d octets", sizeof(inttab));
24:     printf("\nTaille de floattab = %d octets", sizeof(floattab));
25:     printf("\nTaille de doubletab = %d octets\n", sizeof(doubletab));
26:     exit(EXIT_SUCCESS);
27:  }
```

La liste suivante correspond à des programmes UNIX et Windows 32 bits :

```
La taille de int est de 4 octets
La taille de short est de 2 octets
La taille de long est de 4 octets
La taille de float est de 4 octets
La taille de double est de 8 octets
Taille de inttab = 400 octets
Taille de floattab = 400 octets
Taille de doubletab = 800 octets
```

Analyse

Saisissez et compilez ce programme en suivant la procédure du Chapitre 1. Son exécution vous donnera la taille en octets des trois tableaux et des variables numériques.

Les lignes 7, 8 et 9 déclarent trois tableaux qui contiendront des types de données différents. Leur taille respective est alors affichée aux lignes 23 à 25. Elles sont calculées en multipliant la taille de la donnée stockée dans le tableau par le nombre d'éléments du tableau. Exécutez le programme et contrôlez les résultats. Comme vous avez pu le constater dans les résultats précédents, des machines ou des systèmes d'exploitation différents pourront travailler avec des types de données de taille différente.

Résumé

Les tableaux numériques fournissent une méthode puissante de stockage des données. Ils permettent de grouper des données de même type sous un nom de groupe unique. Chaque donnée, ou élément, est identifiée en utilisant un index entre crochets après le nom du tableau. Les tâches de programmation qui impliquent un traitement répétitif des données conduisent naturellement à l'utilisation de tableaux.

Avant d'être utilisé, un tableau doit être déclaré. Il est possible d'initialiser quelques éléments du tableau dans cette déclaration.

Q & R

Q Que se passera-t-il si j'utilise une taille d'index supérieure au nombre d'éléments du tableau ?

R Si l'index ne correspond pas à la déclaration du tableau, le programme sera compilé et pourra même tourner. Les résultats de cette exécution sont imprévisibles et il pourrait être très difficile de retrouver la source des erreurs qui en découleront. Soyez donc très prudent en initialisant et en stockant des données dans un tableau.

Q Peut-on utiliser un tableau sans l'avoir initialisé ?

R Cette erreur n'est pas détectée par le compilateur. Le tableau peut contenir n'importe quoi et le résultat de son utilisation est imprévisible. En l'initialisant, vous connaissez la valeur des données qui y sont stockées.

Q Combien de dimensions un tableau peut-il avoir ?

R La seule limite au nombre de dimensions est imposée par la place mémoire. Les besoins en mémoire d'un tableau augmentent considérablement avec le nombre de dimensions. Il faut éviter de gaspiller la mémoire disponible en déclarant des tableaux ayant juste la taille nécessaire.

Q Comment peut-on initialiser entièrement et facilement un tableau ?

R Vous pouvez le faire avec une instruction de déclaration comme celle que nous avons étudiée dans ce chapitre, ou à l'aide d'une boucle `for`.

Q Quel intérêt y a-t-il à utiliser un tableau plutôt que des variables simples ?

R Dans le cas du tableau, des données de même type sont stockées sous un seul nom. Le programme du Listing 8.3 manipulait 1000 valeurs de données. La création et l'initialisation de 1000 variables différentes sont inconcevables, l'usage d'un tableau a considérablement simplifié le travail.

Q **Que faire lorsque l'on ne peut pas prévoir la taille du tableau lors de l'écriture du programme ?**

R Certaines fonctions en langage C permettent de réserver la mémoire nécessaire pour des variables ou des tableaux de façon dynamique. Ces fonctions seront traitées au Chapitre 15.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Quels types de données peut-on stocker dans un tableau ?
2. Quelle est la valeur d'index du premier des dix éléments d'un tableau ?
3. Quelle est la dernière valeur d'index d'un tableau à une dimension qui contient *n* éléments ?
4. Que se passe-t-il si un programme essaye d'accéder à un élément de tableau avec un index invalide ?
5. Comment faut-il déclarer un tableau à plusieurs dimensions ?
6. Quelle est la taille du tableau suivant ?

```
int tableau[2][3][5][8];
```

7. Comment s'appelle le dixième élément du tableau de la question 6 ?

Exercices

1. Écrivez une ligne de code déclarant trois tableaux à une dimension pour stocker des entiers qui s'appelleraient `un`, `deux`, et `trois` avec 1000 éléments chacun.
2. Écrivez la déclaration d'un tableau de 10 éléments initialisés à 1 pour stocker des entiers.
3. Écrivez le code nécessaire à l'initialisation des éléments du tableau suivant avec la valeur 88 :

```
int huitethuit[88];
```

4. Écrivez le code nécessaire à l'initialisation des éléments du tableau suivant avec la valeur 0 :

```
int stuff[12][10];
```

5. **CHERCHEZ L'ERREUR** :

```
int x, y;
int tableau[10][3];
int main()
{
      for (x = 0; x < 3; x++)
      for (y = 0; y < 10; y++)
      tableau[x][y] = 0;
   exit(EXIT_SUCCESS);
}
```

6. **CHERCHEZ L'ERREUR** :

```
int tableau[10];
int x = 1;

int main()
{
    for (x = 1; x <= 10; x++)
    tableau[x] = 99;

   exit(EXIT_SUCCESS);
}
```

7. Écrivez un programme qui stocke des nombres aléatoires dans un tableau à deux dimensions de 5 par 4. Affichez à l'écran la valeur des éléments du tableau en colonnes (utilisez la fonction `rand()` du Listing 8.3).

8. Modifiez le Listing 8.3 pour utiliser un tableau à une dimension. Calculez et affichez la moyenne des 1000 valeurs avant de les afficher individuellement. N'oubliez pas de mettre le programme en pause après l'affichage d'un groupe de 10 valeurs.

9. Écrivez un programme qui initialise un tableau de 10 éléments. Chaque élément devra avoir la valeur de son index. L'exécution du programme se terminera en affichant le contenu des 10 éléments.

10. Modifiez le programme de l'exercice 9 pour qu'il copie ensuite ses éléments dans un nouveau tableau en ajoutant 10 à chacune des valeurs. Affichez la valeur des éléments du second tableau.

9

Les pointeurs

Les pointeurs jouent un rôle très important dans le langage C. Ils permettent de manipuler les données dans vos programmes. Aujourd'hui, vous allez étudier :

- La définition d'un pointeur
- L'utilisation des pointeurs
- La déclaration et l'initialisation des pointeurs
- L'utilisation des pointeurs avec des variables simples et des tableaux
- Le passage des tableaux à une fonction avec les pointeurs

L'utilisation de pointeurs offre de nombreux avantages qui peuvent être partagés en deux catégories : celle des tâches qui sont exécutées de manière plus simple avec des pointeurs, et celle des tâches qui ne pourraient pas être réalisées sans pointeurs. Pour devenir un bon programmeur en langage C, il est impératif de bien comprendre les principes de fonctionnement des pointeurs.

Définition

Pour comprendre ce que sont les pointeurs, vous devez avoir une idée de la façon dont votre ordinateur stocke les informations en mémoire.

La mémoire de votre ordinateur

La mémoire vive de votre PC est constituée de millions d'emplacements mémoire rangés de façon séquentielle. Chaque emplacement a une adresse unique, comprise entre 0 et une valeur maximale qui dépend de la quantité de mémoire installée sur votre machine.

Quand votre ordinateur fonctionne, une partie de sa mémoire vive est occupée par le système d'exploitation. Si vous lancez un programme, le code (les instructions en langage machine) et les données qu'il utilise occuperont en partie cette mémoire. Nous allons étudier de quelle façon votre programme occupera cette mémoire.

Quand un programme déclare une variable, le compilateur réserve un emplacement mémoire avec une adresse unique pour stocker cette variable. Il associe l'adresse au nom de la variable. Quand le programme utilise le nom de la variable, il accède automatiquement à l'emplacement mémoire correspondant. Ce mécanisme est transparent pour l'utilisateur du programme, qui utilise le nom de la variable sans se soucier de l'endroit où l'ordinateur l'a stockée. Cela est illustré par le schéma de la Figure 9.1.

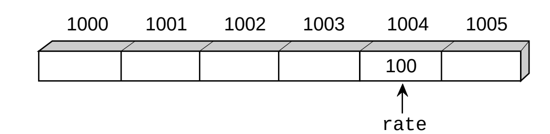


Figure 9.1
Une variable de programme a une adresse de mémoire spécifique.

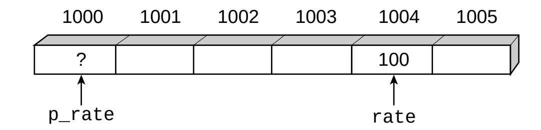
Une variable appelée `rate` est déclarée et initialisée à 100. Le compilateur a réservé un emplacement mémoire à l'adresse 1004, qu'il associe donc au nom de la variable.

Création d'un pointeur

Vous pouvez remarquer que l'adresse de la variable `rate` est un nombre, et qu'elle peut donc être utilisée comme n'importe quel autre nombre en langage C. Si vous connaissez l'adresse d'une variable, vous pouvez créer une autre variable pour y stocker l'adresse de la première. Dans notre exemple, la première étape consiste à déclarer la variable dans laquelle on stockera l'adresse de `rate`. Nous allons l'appeler `p rate`. Le schéma suivant

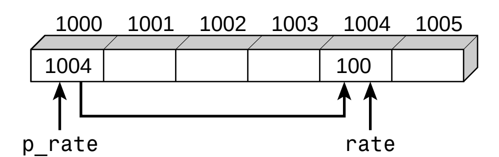
montre qu'un emplacement a bien été réservé pour `p rate`, mais la valeur qui y est stockée est inconnue.

Figure 9.2
Un emplacement a été alloué à la variable p_rate.



L'étape suivante consiste à stocker l'adresse de `rate` dans la variable `p rate`. Celle-ci représente maintenant l'emplacement mémoire de la variable `rate` : en langage C, `p rate` pointe sur `rate`, ou `p rate` est un pointeur vers `rate`.

Figure 9.3
La variable p_rate contient l'adresse de la variable rate, elle est également un pointeur vers rate.



En résumé, un pointeur est une variable qui contient l'adresse d'une autre variable. Étudions maintenant l'utilisation de ces pointeurs dans un programme C.

Pointeurs et variables simples

L'exemple précédent montrait une variable pointant sur une autre variable simple (ce n'est pas un tableau). Voyons maintenant comment créer et utiliser ce type de pointeur.

Déclaration

Un pointeur est une variable numérique qui doit être déclarée, comme toutes les variables, avant d'être utilisée. Le nom des pointeurs suit les mêmes règles que celui des autres variables et doit être unique. Nous utilisons, dans ce chapitre, une convention pour le nom des pointeurs. Si le nom de la variable est `nom`, le pointeur s'appellera `p nom`. Ce n'est pas une règle, vous pouvez choisir votre propre convention.

La déclaration d'un pointeur a la forme suivante :

```
nomtype *nomptr;
```

`nomtype` représente le type de la variable pointée. L'astérisque (`*`) est *l'opérateur indirect*, il indique que `nomptr` est un pointeur vers une variable de type `nomtype`, et non une variable

de type `nomtype`. Une déclaration peut contenir des pointeurs et des variables. Voici quelques exemples :

```
char *ch1, *ch2;     /* ch1 et ch2 pointent sur une variable */
                     /* de type char                         */
float *valeur, pourcent;  /* valeur est un pointeur vers une */
                     /* variable de type float et pourcent   */
                     /* est une variable de type float       */
```



Le symbole `*` représente l'opérateur indirect et l'opérateur de multiplication. Le compilateur fera la différence en fonction du contexte.

Initialisation

Déclarer un pointeur n'est pas suffisant ; si vous ne le faites pas pointer sur une variable, il est inutile. Pour les mêmes raisons qu'avec des variables, travailler avec un pointeur qui n'a pas été initialisé peut se révéler désastreux. Un pointeur doit contenir l'adresse d'une variable, et c'est le programme qui doit s'en charger en utilisant l'opérateur d'adresse (`&`). Quand il est placé avant le nom de la variable, l'opérateur d'adresse renvoie l'adresse de cette variable. L'initialisation d'un pointeur est donc une instruction de la forme :

```
pointeur = &variable;
```

Si nous reprenons l'exemple de la Figure 9.3, l'instruction correspondante aurait été :

```
p_rate = &rate;   /* on attribue l'adresse de rate à p_rate */
```

Avant cette instruction, `p rate` ne pointait vers rien de particulier. Après, `p rate` devient un pointeur vers `rate`.

Pointeurs, mode d'emploi

Maintenant que vous savez déclarer et initialiser un pointeur, vous devez apprendre à l'utiliser. L'opérateur indirect (`*`) entre de nouveau en jeu. Quand cet opérateur précède le nom d'un pointeur, il fait référence à la variable qui est pointée.

Reprenons notre pointeur `p rate` initialisé pour pointer vers la variable `rate`. `*p rate` représente la variable `rate`. Pour afficher la valeur de la variable, vous pouvez écrire :

```
printf("%d", rate),
```

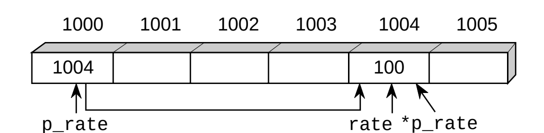
ou bien

```
printf("%d", *p_rate);
```

En langage C, ces deux instructions sont équivalentes. Si vous accédez au contenu de la variable en utilisant son nom, vous effectuez un *accès direct*. Si vous accédez au contenu d'une variable par l'intermédiaire de son pointeur, vous effectuez un *accès indirect* ou une *indirection*.

La Figure 9.4 montre que le nom d'un pointeur précédé d'un opérateur d'indirection se réfère à la valeur pointée.

Figure 9.4
L'opérateur d'indirection associé à un pointeur.



Les pointeurs sont très importants en langage C, il est essentiel de bien comprendre leur fonctionnement. Si votre pointeur s'appelle `ptr` et qu'il a été initialisé pour pointer sur la variable `var` alors :

- `*ptr` et `var` représentent le contenu de `var`.
- `ptr` et `&var` représentent l'adresse de `var`.

Listing 9.1 : Utilisation des pointeurs

```
/* Exemple simple d'utilisation d'un pointeur. */
#include <stdio.h>
#include <stdlib.h>

/* Déclaration et initialisation d'une variable int */

int var = 1;

/* Déclaration d'un pointeur vers une variable int */

int *ptr;

int main()
{
   /* Initialisation de ptr  */

   ptr = &var;

   /* Accès direct et indirect à var */

```

Listing 9.1 : Utilisation des pointeurs *(suite)*

```
21:     printf("Accès direct, var = %d\n", var);
22:     printf("Accès indirect, var = %d\n\n", *ptr);
23:
24:     /* Affichage de l'adresse avec les deux méthodes */
25:
26:     printf("L'adresse de var = %d\n", &var);
27:     printf("L'adresse de var = %d\n", ptr);
28:
29:     exit(EXIT_FAILURE);
30:  }
```



```
Accès direct, var = 1
Accès indirect, var = 1

L'adresse de var = 4264228
L'adresse de var = 4264228
```



Sur votre ordinateur, l'adresse de la variable `var` sera certainement différente de 4264228.

Analyse

Ce programme utilise deux variables. La ligne 7 déclare la variable `var` de type `int` et l'initialise à 1. La ligne 11 contient la déclaration du pointeur `ptr` vers une variable de type `int`. La ligne 17 attribue l'adresse de `var` au pointeur avec l'opérateur d'adresse (`&`). Le programme affiche ensuite la valeur de ces deux variables à l'écran. La ligne 21 affiche la valeur de `var`, et la ligne 22 affiche la valeur stockée à l'adresse sur laquelle `ptr` pointe. Dans notre exemple, cette valeur est 1. La ligne 26 affiche l'adresse de `var` en utilisant l'opérateur d'adresse. La ligne 27 affiche la même adresse en utilisant le pointeur `ptr`.

Ce programme illustre bien les relations qui existent entre une variable, son adresse, un pointeur et la référence à la variable pointée.



À faire

Veillez à bien comprendre ce que représente un pointeur et comment il travaille. Le langage C et les pointeurs vont de pair.

À ne pas faire

N'utilisez pas un pointeur qui n'a pas été initialisé. Les résultats pourraient être désastreux.

Pointeurs et types de variables

Les différents types de variables du langage C n’occupent pas tous la même mémoire. Sur la plupart des systèmes, une variable `int` prend quatre octets, une variable `double` en prend huit, etc. Chaque octet en mémoire possède sa propre adresse ; une variable qui occupe plusieurs octets occupe donc plusieurs adresses.

L’adresse d’une donnée est en fait l’adresse du premier octet occupé. Voici un exemple qui déclare et initialise trois variables :

```
int vint = 12252;
char vchar = 90;
double vdouble = 1200.156004;
```

Ces variables sont stockées en mémoire comme le montre la Figure 9.5. La variable `int` occupe quatre octets, la variable `char` en occupe un, et la variable `double` huit.

Voici la déclaration et l’initialisation des pointeurs vers ces trois variables :

```
int *p_vint;
char *p_vchar;
double *p_vdouble;
/* des instructions peuvent être insérées ici */
p_vint = &vint;
p_vchar = &vchar;
p_vdouble = &vdouble;
```

Chaque pointeur contient l’adresse du premier octet de la variable pointée. Ainsi, `p vint` est égal à 1000, `p vchar` a la valeur 1005 et `p vdouble` est égal à 1008. Le compilateur sait qu’un pointeur vers une variable de type `int` pointe sur le premier des quatre octets, qu’un pointeur de variable de type double pointe sur le premier des huit octets, etc.

Les Figures 9.5 et 9.6 représentent les trois variables de l’exemple, séparées par des emplacements vides. Leur seul intérêt est de rendre la figure plus lisible ; dans la réalité, le compilateur aurait stocké les trois variables dans des emplacements mémoire adjacents.

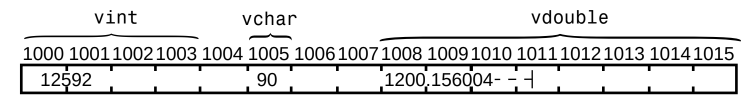


Figure 9.5
Les différents types de variables n’occupent pas tous la même quantité de mémoire.

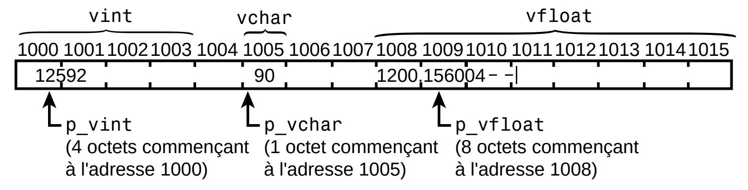


Figure 9.6
Le compilateur "connaît" la taille des variables vers lesquelles pointe un pointeur.

Pointeurs et tableaux

Les pointeurs sont très utiles pour travailler avec les variables, mais ils le sont encore plus quand on les utilise avec les tableaux. En fait, les index de tableaux dont on a parlé au Chapitre 8 ne sont rien d'autre que des pointeurs.

Noms de tableau et pointeurs

Un nom de tableau sans les crochets s'utilise la plupart du temps comme un pointeur vers le premier élément du tableau. Si vous avez déclaré le tableau `data[]`, `data` aura la valeur de l'adresse de `data[0]` et sera donc équivalent à l'expression `&data[0]`.



Le nom du tableau n'est pas un pointeur même s'il est souvent utilisé comme tel. Par exemple, contrairement à un pointeur, il n'est pas possible de modifier la valeur d'un tableau. Un autre exemple est la valeur de `sizeof()` qui, pour un pointeur, renvoie toujours la même valeur (même s'il pointe sur un tableau) alors que pour un tableau, il renvoie une valeur qui dépend de la taille du tableau.

Vous pouvez quand même déclarer un pointeur variable et l'initialiser pour pointer sur le premier élément du tableau :

```
int tableau[100], *p_tableau;
/* instructions */
p_tableau = tableau;
```

`p tableau` étant un pointeur variable, il peut être modifié pour pointer ailleurs. Contrairement à `tableau`, `p tableau` ne pointe pas obligatoirement sur le premier élément de `tableau[]`. Il pourrait, par exemple, pointer vers d'autres éléments du tableau. Mais avant cela, il faut savoir comment sont stockés en mémoire les éléments d'un tableau.

Stockage des éléments d'un tableau

Les éléments d'un tableau sont stockés dans des emplacements mémoire séquentiels, le premier élément ayant l'adresse la plus basse. L'adresse d'un autre élément, par rapport au premier, dépend de la taille des valeurs stockées dans le tableau et de l'index de cet élément.

Prenons comme exemple un tableau de type `int`. Une variable `int` occupe quatre octets en mémoire. Chaque élément du tableau sera donc situé quatre octets plus loin que le précédent, et

son adresse sera égale à celle de l'élément précédent plus quatre. Avec des variables de type double (8 octets), chaque élément serait stocké dans huit octets adjacents ; la différence entre les adresses de deux éléments voisins serait alors de huit.

La Figure 9.7 illustre les relations existant entre le stockage du tableau et les adresses, pour un tableau de type `int` avec six éléments, et pour un tableau de type `double` avec trois éléments.

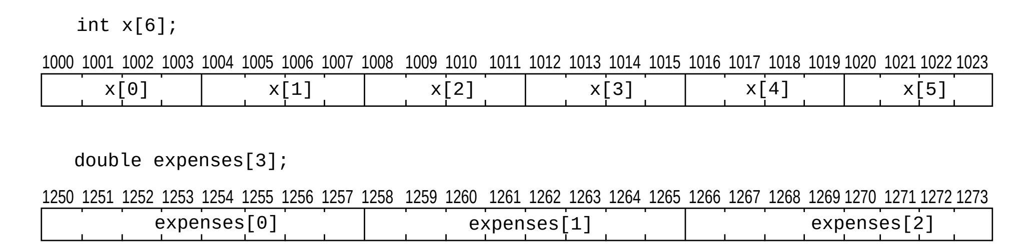


Figure 9.7
Stockage de tableaux contenant différents types de données.

En étudiant la Figure 9.7, vous comprendrez pourquoi les expressions suivantes sont vraies :

```
1:  x == 1000
2:  &x[0] == 1000
3:  &x[1] == 1004
4:  expenses == 1250
5:  &expenses[0] == 1250
6:  &expenses[1] == 1258
```

`x` exprimé sans crochets est l'adresse du premier élément (`x[0]`). La figure nous montre que `x[0]` se situe à l'adresse 1000, ce qui justifie aussi la ligne 2. La ligne 3 indique que l'adresse du second élément (index 1) est 1004. Il est facile de le vérifier sur la figure. Les lignes 4, 5, et 6 sont équivalentes aux lignes 1, 2, et 3 respectivement. La différence tient aux adresses qui vont de quatre en quatre dans le cas des données `int`, et de huit en huit pour les données de type `double`.

Vous pouvez constater, à partir de ces exemples, qu'un pointeur devra être incrémenté de quatre pour accéder à des éléments successifs d'un tableau de type `int`, et de huit dans le cas d'un tableau de type `double`.

Pour accéder à des éléments successifs d'un tableau contenant un type de donnée particulier, le pointeur devra être incrémenté de la valeur `sizeof(typedonnée)`. L'opérateur `sizeof` renvoie la taille en octets du type de donnée reçu en argument. Mieux, s'il s'agit bien d'un pointeur et non d'un tableau, incrémentez-le de la valeur de `sizeof(ptr)` où `ptr` est le nom de votre pointeur.

Listing 9.2 : Affichage des adresses d'éléments successifs d'un tableau

```
/* Ce programme vous montre la relation existant entre les   */
/* adresses et les éléments de tableaux contenant différents */
/* types de données.                                         */

#include <stdio.h>
#include <stdlib.h>

/* Déclaration de trois tableaux et d'une variable compteur. */
short sh[10], x;
int i[10];
double d[10];

int main()
{
    /* Affichage de l'en-tête du tableau */

    printf("\t\tShort\t\tEntier\t\tDouble");

    printf("\n================================");
    printf("======================");

    /* Affichage de l'adresse de chaque élément du tableau. */

    for (x = 0; x < 10; x++)
           printf("\nElement %d:\t%ld\t\t%ld\t\t%ld", x,
                  &sh[x],&i[x], &d[x]);

       printf("\n================================");
       printf("======================\n");
       exit(EXIT_SUCCESS);
}
```

```
              Short      Entier      Double
================================================
Element 0:    1392         1414        1454
Element 1:    1394         1418        1462
Element 2:    1396         1422        1470
Element 3:    1398         1426        1478
Element 4:    1400         1430        1486
Element 5:    1402         1434        1494
Element 6:    1404         1438        1502
Element 7:    1406         1442        1510
Element 8:    1408         1446        1518
Element 9:    1410         1450        1526
================================================
```

Sur votre système, les adresses seront différentes, mais la différence entre deux adresses sera identique. Il y a deux octets entre deux éléments de type `short`, quatre octets entre chaque élément de type int et 8 octets entre deux éléments `double`.

Analyse

Ce programme utilise le caractère (`\`) que nous avons étudié au Chapitre 7 pour mettre en forme le tableau qui sera affiché. La fonction `printf()`, appelée en lignes 16 et 24, utilise (`\t`) pour aligner les colonnes du tableau.

Les trois tableaux du programme sont déclarés aux lignes 8, 9 et 10. Le tableau `sh` est de type short, `i` est de type `int` et `d` de type `double`. La ligne 16 affiche l'en-tête des colonnes et les lignes 18, 19, 27 et 28 des signes (=) pour séparer les résultats. La boucle `for`, en lignes 23, 24 et 25, affiche chaque ligne de résultats.

Pointeur arithmétique

Nous venons de voir que le pointeur du premier élément d'un tableau doit être incrémenté d'un nombre d'octets égal à la taille des données du tableau pour pointer sur l'élément suivant. Pour pointer sur un élément quelconque en utilisant une notation de type pointeur, on utilise le *pointeur arithmétique*.

Incrémenter les pointeurs

Incrémenter un pointeur consiste à en augmenter la valeur. Si vous incrémentez un pointeur de 1, le pointeur arithmétique va augmenter sa valeur pour qu'il accède à l'élément de tableau suivant. En fait, C connaît le type de donnée du tableau à partir de la déclaration, et il va incrémenter le pointeur de la taille de cette donnée chaque fois.

Par exemple, si `ptr int` pointe sur un élément de tableau de type `int`, l'instruction suivante :

```
ptr_int++;
```

incrémente la valeur de ce pointeur de 4 pour qu'il pointe sur l'élément `int` suivant.

De la même façon, si vous augmentez la valeur du pointeur de *n*, C va incrémenter ce pointeur pour qu'il pointe sur le *n-ième* élément suivant :

```
ptr_int += 2;
```

Cette instruction va augmenter de 8 la valeur du pointeur, pour qu'il pointe 2 éléments plus loin.

Décrémenter les pointeurs

La décrémentation des pointeurs suit le même principe que l'incrémentation. Si vous utilisez les opérateurs (—) ou (–=) pour décrémenter un pointeur, le pointeur arithmétique va diminuer sa valeur automatiquement en fonction de la taille des données pointées.

Le Listing 9.3 présente un exemple d'utilisation du pointeur arithmétique pour accéder aux éléments d'un tableau. L'incrémentation du pointeur permet au programme de se déplacer facilement dans le tableau.

Listing 9.3 : Utilisation d'un pointeur arithmétique pour accéder aux éléments d'un tableau

```
/* Utilisation d'un pointeur arithmétique pour accéder */
/* aux éléments d'un tableau. */
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

/* Déclaration et initialisation d'un tableau d'entiers. */

int i_tableau[MAX] = { 0,1,2,3,4,5,6,7,8,9 };

/* Déclaration d'un pointeur vers int et d'une variable int. */

int *i_ptr, count;

/* Déclaration et initialisation d'un tableau de type double. */

double d_tableau[MAX] = {.0, .1, .2, .3, .4, .5, .6, .7, .8, .9};

/* Déclaration d'un pointeur vers double. */

double *d_ptr;

int main()
{
/* Initialisation des pointeurs. */

   i_ptr = i_tableau;
   d_ptr = d_tableau;

/* Affichage des éléments du tableau. */

   for (count = 0; count < MAX; count++)
     printf("%d\t%f\n", *i_ptr++, *d_ptr++);

   exit(EXIT_SUCCESS);
}
```

```
0    0.000000
1    0.100000
2    0.200000
3    0.300000
4    0.400000
5    0.500000
```

```
6    0.600000
7    0.700000
8    0.800000
9    0.900000
```

Analyse

La ligne 5 définie et initialise à 10 la constante `MAX` qui sera utilisée tout au long de ce programme. En ligne 9, `MAX` indique le nombre d'entiers stockés dans `i tableau`. La ligne 13 déclare un pointeur appelé `i ptr` et une variable simple `count` de type `int`. Un second tableau, de type `double`, est défini et initialisé en ligne 17. Ce tableau contient aussi `MAX` éléments. La ligne 21 déclare le pointeur `d ptr` vers les données double.

La fonction `main()` commence en ligne 23 et finit en ligne 36. Le programme attribue l'adresse de début des deux tableaux à leur pointeur respectif en lignes 27 et 28. L'instruction `for`, en lignes 32 et 33, utilise la variable `count` pour s'exécuter `MAX` fois. À chaque exécution, la ligne 33 affiche le contenu des éléments pointés, avec la fonction `printf()`, puis incrémente les deux pointeurs pour accéder aux deux éléments de tableau suivants.

L'utilisation de pointeurs arithmétiques dans le Listing 9.3 n'offre pas d'avantages particuliers. L'utilisation de l'index aurait été plus simple. Quand vous commencerez à écrire des programmes plus complexes, vous trouverez très vite l'emploi de ce type de pointeur avantageux.

Souvenez-vous qu'il n'est pas possible d'incrémenter ou de décrémenter un *pointeur constant* (un nom de tableau sans les crochets). Souvenez-vous aussi que lorsque vous vous déplacez dans un tableau à l'aide d'un pointeur, le compilateur C ne garde pas de trace du début et de la fin du tableau. Les données qui sont stockées avant ou après le tableau ne sont pas des éléments ; soyez donc très prudent et contrôlez l'emplacement des données pointées.

Autre utilisation des pointeurs

La dernière opération que l'on peut effectuer avec des pointeurs est la *différence* entre deux pointeurs. Si vous avez deux pointeurs sur un même tableau, le résultat de leur soustraction correspond au nombre d'éléments les séparant. Exemple :

```
ptr1 - ptr2
```

Cette instruction donne le nombre d'éléments qui séparent les deux éléments pointés par `ptr1` et `ptr2`. Les opérateurs de comparaison ==, !=, >, <, >= et <= peuvent aussi être

utilisés. Les premiers éléments d'un tableau ont toujours une adresse plus basse que les derniers. Ainsi, si `ptr1` et `ptr2` sont deux pointeurs d'un même tableau, la comparaison :

```
ptr1 < ptr2
```

est vraie si `ptr1` pointe sur un élément d'index plus petit que `ptr2`.

Beaucoup d'opérations arithmétiques sont réservées aux variables simples, car elles n'auraient aucun sens avec les pointeurs. Par exemple, si `ptr` est un pointeur, l'instruction :

```
ptr *= 2;
```

générera un message d'erreur. Le Tableau 9.1 vous donne la liste des six opérations possibles avec les pointeurs.

Tableau 9.1 : Opérations sur les pointeurs

Opérateur	*Description*
Affectation	Vous pouvez attribuer une valeur à un pointeur. Cette valeur doit correspondre à une adresse obtenue avec l'opérateur d'adresse (&,), ou à partir d'un pointeur constant.
Indirection	L'opérateur indirect (*) donne la valeur stockée à l'emplacement pointé.
Adresse de	Vous pouvez utiliser l'opérateur d'adresse pour trouver l'adresse d'un pointeur et obtenir un pointeur vers un pointeur.
Incrément	On peut ajouter un nombre entier à la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.
Décrément	On peut soustraire un entier à la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.
Différence	Vous pouvez soustraire un entier de la valeur d'un pointeur pour pointer sur un emplacement mémoire différent.
Comparaison	Ces opérateurs ne sont valides que pour deux pointeurs d'un même tableau.

Précautions d'emploi

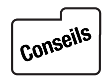
Quand vous utilisez des pointeurs dans un programme, une grosse erreur est à éviter : utiliser un pointeur non initialisé à gauche d'une instruction d'affectation. Par exemple, dans la déclaration suivante :

```
int *ptr;
```

le pointeur n'est pas initialisé. Cela signifie qu'il ne pointe pas sur un emplacement *connu*. Si vous écrivez :

```
*ptr = 12;
```

la valeur 12 va être stockée à l'adresse (inconnue) pointée par `ptr`. Cette adresse peut se situer n'importe où en mémoire, au milieu du code du système d'exploitation par exemple. La valeur 12 risque d'écraser une donnée importante, et le résultat peut aller de simples erreurs dans un programme à l'arrêt complet du système.



À ne pas faire

Effectuer des opérations mathématiques comme des divisions, des multiplications ou des modulos avec des pointeurs. Les seules opérations possibles sont l'incrémentation ou le calcul de la différence entre deux pointeurs d'un même tableau.

N'oubliez pas que l'ajout ou la soustraction d'un entier à un pointeur change la valeur de ce pointeur en fonction de la taille des données sur lesquelles il pointe.

Incrémenter ou décrémenter une variable de tableau. Initialisez un pointeur avec l'adresse de début du tableau et incrémentez-le.

À faire

Renseignez-vous sur la taille des différents types de données dans votre ordinateur.

Pointeurs et index de tableaux

Un nom de tableau sans les crochets est un pointeur vers le premier élément du tableau. Par conséquent, vous pouvez accéder au premier élément de ce tableau avec l'opérateur indirect (`*`). Si `tab[]` est un tableau, l'expression `*tab` représente le premier élément de ce tableau, `*(tab+1)` est le deuxième élément, etc. En généralisant, nous obtenons les relations suivantes :

```
*(tab) == tab[0]
*(tab+1) == tab[1]
*(tab+2) == tab[2]
etc.
*(tab+n) == tab[n]
```

Ces expressions vous donnent les équivalences entre la notation de l'index et celle utilisant les pointeurs. Le compilateur C ne fait aucune différence entre ces deux méthodes d'accès aux données d'un tableau.

Passer des tableaux à une fonction

Pointeurs et tableaux sont étroitement liés en langage C, et cette relation va permettre le passage d'un tableau comme argument d'une fonction.

Comme nous l'avons appris au Chapitre 5, un argument est une valeur passée à une fonction par le programme appelant. Ce doit être une valeur numérique de type `int`, `float` ou autre. Un élément de tableau peut être transmis à une fonction, mais pas un tableau tout entier. Un pointeur étant une valeur numérique (adresse), vous pouvez passer cette valeur à une fonction. La fonction connaissant l'adresse du tableau, elle pourra accéder à tous ses éléments en utilisant un pointeur.

Si vous passez un tableau en argument à une fonction, un problème va se poser. Si la fonction doit accéder à différents éléments (par exemple, trouver l'élément ayant la plus grande valeur) de ce tableau, elle doit en connaître le nombre. Il y a deux méthodes pour faire connaître la taille du tableau à la fonction.

Vous pouvez identifier le dernier élément d'un tableau en y stockant une valeur particulière. Quand la fonction accédera aux éléments du tableau elle reconnaîtra le dernier par cette valeur. L'inconvénient de cette méthode est que vous devez réserver une valeur pour l'indicateur de fin de tableau. Vous n'êtes plus libre de stocker toutes les valeurs possibles dans votre tableau.

L'autre méthode est plus directe. On passe la taille du tableau en argument. La fonction en reçoit donc deux, le pointeur sur le premier élément et un nombre entier indiquant le nombre d'éléments du tableau. C'est celle que nous avons choisie dans ce livre. Ce n'est pas forcément la meilleure : cela dépend des cas.

Le Listing 9.4 présente un programme qui lit une série de valeurs entrées par l'utilisateur et la stocke dans un tableau. Il appelle ensuite la fonction `largest()` en lui passant le tableau (pointeur et taille). La fonction cherche la plus grande valeur stockée dans le tableau et la renvoie au programme.

Listing 9.4 : Exemple de passage d'un tableau à une fonction

```
/* Comment passer d'un tableau à une fonction. */
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

int tab[MAX], count;

int largest(int x[], int y);

int main()
```

```
12:  {
13:      /* Lecture des MAX valeurs à partir du clavier. */
14:
15:      for (count = 0; count < MAX; count++)
16:      {
17:      printf("Entrez une valeur entière :  ");
18:      scanf("%d", &tab[count]);
19:      }
20:
21:      /* Appel de la fonction et affichage de la valeur renvoyée. */
22:
23:      printf("\n\nLa valeur la plus grande est %d\n",
              largest(tab, MAX));
24:      exit(EXIT_SUCCESS);
25:  }
26:
27:  /* La fonction largest() renvoie la valeur la plus grande */
28:  /* d'un tableau d'entiers. */
29:
30:  int largest(int x[], int y)
31:  {
32:      int count, biggest = x[0];
33:
34:      for (count = 1; count < y; count++)
35:      {
36:          if (x[count] > biggest)
37:          biggest = x[count];
38:      }
39:
40:      return biggest;
41:  }
```

```
Entrez une valeur entière : 1
Entrez une valeur entière : 2
Entrez une valeur entière : 3
Entrez une valeur entière : 4
Entrez une valeur entière : 5
Entrez une valeur entière : 10
Entrez une valeur entière : 9
Entrez une valeur entière : 8
Entrez une valeur entière : 7
Entrez une valeur entière : 6

La valeur la plus grande est 10
```

Analyse

Les lignes 9 et 30 contiennent le prototype et l'en-tête de la fonction `largest()`. Le premier argument passé, `int x[]`, est un pointeur de type `int`. Le deuxième, `y`, est un entier. La déclaration de cette fonction aurait pu s'écrire :

```
int largest(int *x, int y);
```

Les deux formes sont équivalentes : `int x[]` et `int *x` signifient "pointeur vers une donnée entière".

Au moment de l'appel de la fonction `largest()`, `x` contient la valeur du premier argument, c'est donc un pointeur vers le premier élément du tableau. Vous pouvez utiliser `x` partout où un pointeur peut être utilisé. Dans cette fonction, on accède aux éléments du tableau au moyen de l'index (lignes 36 et 37). La notation "pointeur" aurait pu être utilisée de cette façon :

```
for (count = 0; count < y; count++)
{
    if (*(x+count) > biggest)
        biggest = *(x+count);
}
```

Le Listing 9.5 présente une autre méthode pour passer d'un tableau à une fonction.

Listing 9.5 : Une autre méthode pour passer d'un tableau à une fonction

```
/*   Comment transmettre un tableau à une fonction. Autre méthode.  */
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

int tab[MAX+1], count;

int largest(int x[]);

int main()
{
/* Lecture des MAX valeurs à partir du clavier. */

    for (count = 0; count < MAX; count++)
    {
        printf("Entrez une valeur entière :  ");
        scanf("%d", &tab[count]);

        if (tab[count] == 0)
             count = MAX;         /* sortie de la boucle */
    }
    tab[MAX] = 0;

/* Appel de la fonction et affichage de la valeur renvoyée. */

    printf("\n\nLa valeur la plus grande est %d\n", largest(tab));
    exit(EXIT_SUCCESS);
}

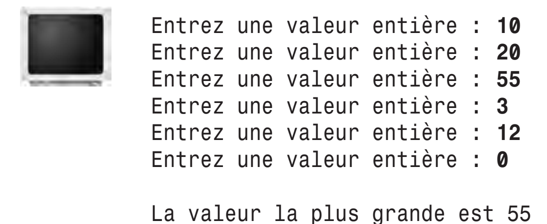
/* La fonction largest() renvoie la plus grande valeur du tableau. */
```

```
32:
33:  int largest(int x[])
34:  {
35:  int count, biggest = x[0];
36:
37:      for (count = 1; x[count] != 0; count++)
38:      {
39:          if (x[count] > biggest)
40:              biggest = x[count];
41:      }
42:
43:      return biggest;
44:  }
```

```
Entrez une valeur entière : 1
Entrez une valeur entière : 2
Entrez une valeur entière : 3
Entrez une valeur entière : 4
Entrez une valeur entière : 5
Entrez une valeur entière : 10
Entrez une valeur entière : 9
Entrez une valeur entière : 8
Entrez une valeur entière : 7
Entrez une valeur entière : 6

La valeur la plus grande est 10
```

Voici le résultat obtenu après avoir déroulé le même programme une seconde fois :



```
Entrez une valeur entière : 10
Entrez une valeur entière : 20
Entrez une valeur entière : 55
Entrez une valeur entière : 3
Entrez une valeur entière : 12
Entrez une valeur entière : 0

La valeur la plus grande est 55
```

Analyse

La fonction `largest()` de ce programme a le même rôle que celle du code source précédent, mais on ne lui passe que le pointeur. La boucle `for` de la ligne 37 recherche la valeur la plus grande jusqu'à ce quelle trouve la valeur 0. Cette valeur, qui provoque la sortie de la boucle, correspond à la fin du tableau.

Le Listing 9.5 présente quelques différences par rapport au Listing 9.4. La ligne 7, par exemple, ajoute au tableau un élément qui servira d'indicateur de fin. En lignes 20 et 21, une instruction `if` supplémentaire vérifie les données entrées par l'utilisateur, la valeur 0 entraînant la fin de la lecture des données. Si l'utilisateur tape 0, La valeur maximum est

stockée dans la variable `count` pour que la sortie de la boucle `for` se fasse normalement. La ligne 23 initialise le dernier élément à 0.

En ajoutant quelques commandes à la lecture des données, la fonction `largest()` pourrait travailler avec des tableaux de n'importe quelle taille. Il ne faudra pas oublier de mettre un 0 dans le dernier élément, sinon la fonction continuera à lire les données en mémoire au-delà du tableau jusqu'à ce quelle trouve une valeur 0.

Le passage d'un tableau en argument à une fonction n'est pas particulièrement compliqué. Il suffit de transmettre le pointeur du premier élément et, la plupart du temps, le nombre d'éléments du tableau. La fonction pourra ensuite accéder aux différents éléments en utilisant la notation index ou pointeur.



Quand une variable simple est passée à une fonction, c'est une copie de la valeur de cette variable qui est transmise (voir Chapitre 5). La fonction peut utiliser cette valeur, mais elle ne peut pas la changer, car elle n'a pas accès à la variable. Transmettre un tableau à une fonction est un problème différent. La fonction pointe directement sur les éléments du tableau (ce ne sont pas des copies) parce qu'elle en connaît l'adresse. Cette fonction peut donc modifier les valeurs stockées dans le tableau.

Résumé

Les pointeurs constituent un élément important de la programmation en langage C. Un pointeur est une variable qui contient l'adresse d'une autre variable : il "pointe" sur cette variable. Les deux opérateurs liés aux pointeurs sont l'opérateur d'adresse (`&`) et l'opérateur indirect (`*`). L'opérateur d'adresse renvoie l'adresse de la variable devant laquelle il est placé (ex `&var`). L'opérateur indirect renvoie l'adresse de la variable pointée par le pointeur devant lequel il est placé (ex `*ptr`).

Les tableaux et les pointeurs sont aussi liés. Un nom de tableau sans crochets peut être assimilé à un pointeur sur le premier élément du tableau. Les pointeurs arithmétiques permettent d'accéder facilement aux éléments d'un tableau en utilisant la notation pointeur. La notation index est, en fait, une forme de notation pointeur.

Un tableau peut être passé en argument à une fonction par l'intermédiaire de son pointeur. Quand la fonction connaît l'adresse et la taille du tableau, elle peut accéder librement aux éléments de ce tableau en utilisant la méthode index ou la méthode pointeur.

Q & R

Q Pourquoi les pointeurs sont-ils importants en langage C ?

R Les pointeurs vous aident à contrôler le programme et les données. Utilisés avec les fonctions, ils permettent de changer la valeur des données transmises en argument. Le Chapitre 15 vous donnera d'autres exemples d'applications pour les pointeurs.

Q Comment le compilateur sait-il si l'opérateur (`*`) est utilisé pour un calcul de multiplication, une indirection ou une déclaration de pointeur ?

R Le compilateur va interpréter l'astérisque en fonction du contexte. Si l'instruction commence par un type de variable, l'astérisque servira à déclarer un pointeur. Si l'astérisque est utilisé avec une variable déclarée en pointeur dans une instruction qui n'est pas une déclaration de variable, l'astérisque représente une indirection. Si l'astérisque se trouve dans une expression mathématique, sans variable pointeur, elle représente l'opérateur de multiplication.

Q Quel est le résultat de l'utilisation de l'opérateur d'adresse avec un pointeur ?

R Vous obtenez l'adresse du pointeur. Un pointeur n'est qu'une variable qui contient l'adresse de la variable sur laquelle il pointe.

Q Les variables sont-elles toujours stockées au même emplacement mémoire ?

R À chaque exécution d'un programme, ses variables seront stockées à des adresses différentes. Vous ne devez pas attribuer une adresse constante à un pointeur.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Quel opérateur faut-il utiliser pour obtenir l'adresse d'une variable ?
2. Quel opérateur faut-il utiliser pour obtenir la valeur de la variable pointée ?
3. Qu'est-ce qu'un pointeur ?
4. Qu'est-ce qu'un accès indirect ?
5. Comment les éléments d'un tableau sont-ils stockés en mémoire ?
6. Trouvez deux méthodes pour obtenir l'adresse du premier élément du tableau `data[]`.

7. Si on passe un tableau à une fonction, quelles sont les deux méthodes qui permettent d'indiquer la fin du tableau à cette fonction ?
8. Quelles sont les six opérations que l'on peut faire sur des pointeurs ?
9. Supposons que l'on ait deux pointeurs. Le premier pointe sur le troisième élément d'un tableau d'entiers, et le second sur le quatrième élément. Quelle valeur obtiendrez-vous en soustrayant le premier pointeur du second ? (Dans ce cas, la taille d'un entier sera de 4 octets.)
10. Si le tableau de la question 9 contient des données de type `double`, quel sera le résultat de la soustraction ? (En supposant que la taille d'une donnée double soit de 8 octets.)

Exercices

1. Écrivez la déclaration du pointeur `ptr char` pour une variable de type `char`.
2. Soit la variable `cout` de type `int`. Comment déclarer et initialiser le pointeur `p cout` sur cette variable ?
3. Comment peut-on attribuer la valeur 100 à la variable `cout` de la question précédente en utilisant les deux types d'accès : direct et indirect ?
4. En continuant les deux exercices précédents, comment peut-on afficher les valeurs du pointeur et de la variable pointée ?
5. Écrivez l'instruction qui attribue l'adresse de la variable `radius` de type `float` à un pointeur.
6. Trouvez deux méthodes pour attribuer la valeur 100 au troisième élément du tableau `data[]`.
7. Écrivez la fonction `somtabs()` qui, recevant deux tableaux en argument, additionne la valeur des éléments des deux tableaux, puis renvoie le résultat au programme appelant.
8. Écrivez un programme simple utilisant la fonction de l'exercice 7.
9. Écrivez la fonction `addtabs()` qui recevra deux tableaux de même taille. Elle devra additionner les éléments correspondants des deux tableaux, et placer le résultat de la somme dans un troisième tableau de même taille.
10. **TRAVAIL PERSONNEL** : Modifiez la fonction de l'exercice 9 pour qu'elle renvoie le pointeur du tableau contenant les résultats. Placez cette fonction dans un programme qui affichera les valeurs des trois tableaux.

Exemple pratique 3

Une pause

Nous abordons la troisième section de ce type. Vous savez que l'objectif en est de présenter un programme complet plus fonctionnel que les exemples des chapitres. Ce programme comporte très peu d'éléments inconnus, vous n'aurez donc aucune difficulté à le comprendre. Prenez le temps de tester ce code en le modifiant éventuellement et en observant les résultats. Attention aux fautes de frappe qui ne manqueront pas de provoquer des erreurs de compilation.

Listing Exemple pratique 3 : secondes.c

```
/* secondes.c */
/* Programme qui fait une pause. */

 #include <stdio.h>
 #include <stdlib.h>
 #include <time.h>

 void mon_sleep( int nbr_seconds );

int main( void )
{
    int x;
    int wait = 13;

    /* Pause pendant un nombre de secondes déterminé. On affiche *
     * un point pour chaque seconde de pause.                   */

    printf("Pause pendant %d secondes\n", wait );
    printf(">");

```

```
21:     for (x=1; x <= wait; x++)
22:     {
23:        printf(".");       /* affichage d'un point */
24:        fflush(stdout);  /* on force l'affichage du point sur les*/
25:                       /* machines qui utilisent la mémoire tampon*/
26:        mon_sleep( 1 );  /* pause d'une seconde */
27:     }
28:     printf( "Fin !\n");
29:     exit(EXIT_SUCCESS);
30: }
31: /* Pause pendant un nombre de secondes déterminé*/
32: void mon_sleep( int nbr_seconds )
33: {
34:     clock_t goal;
35:
36:     goal = ( nbr_seconds * CLOCKS_PER_SEC ) +  clock();
37:
38:     while( goal > clock() )
39:     {
40:        ; /* loop */
41:     }
42: }
```

Analyse

Ce listing utilise la fonction `mon sleep()` (par similarité avec la fonction système `sleep()` que vous pourrez reprendre dans vos travaux de programmation). Elle permet de mettre en pause l'ordinateur pendant un temps déterminé. La seule activité de ce dernier pendant la pause est d'en contrôler la durée. Cette fonction, ou une de ses variantes, a de nombreuses applications. À cause de la vitesse d'exécution des machines, vous aurez souvent besoin, par exemple, d'introduire une pause pour que l'utilisateur ait le temps de lire les informations présentées à l'écran (affichage d'un copyright à la première exécution d'une application).

Pour illustrer ce processus, le programme affiche un point après chaque pause d'une seconde obtenue avec la fonction `mon sleep()`. Vous pouvez vous amuser à augmenter la durée de cette pause pour contrôler la "précision" de votre ordinateur avec un chronomètre.

Vous pouvez aussi transformer ce programme pour imprimer des points (ou toute autre valeur) pendant un certain temps. Remplacez la ligne 38 par la ligne suivante :

```
printf("x");
```

10

Caractères et chaînes

Un *caractère* peut être une lettre, un chiffre, une marque de ponctuation ou tout autre symbole. Une *chaîne* est une séquence de caractères qui permet de manipuler des textes. Aujourd'hui, vous allez apprendre à :

- Utiliser le type de donnée `char` pour travailler avec des caractères
- Créer des tableaux de type `char` pour stocker des chaînes de caractères
- Initialiser les caractères et les chaînes
- Utiliser les pointeurs avec les chaînes
- Lire et imprimer des caractères ou des chaînes

Le type de donnée *char*

En langage C, `char` est le type de donnée permettant de stocker des caractères. Nous avons vu, au Chapitre 3, que `char` fait partie des types de données numériques.

Le choix de ce type numérique pour stocker des caractères est lié à la façon dont le langage C stocke ses caractères. La mémoire de l'ordinateur conserve toutes les données sous forme numérique. Il n'existe pas de méthode pour stocker directement des caractères. Chaque caractère possède son équivalent en code numérique : c 'est le code ASCII (*American Standart Code for Information Interchange*). Ce code attribue les valeurs 0 à 255 aux lettres majuscules et minuscules, aux chiffres, aux marques de ponctuation et autres symboles. Vous trouverez en Annexe `A`, la totalité de ce code.

Par exemple, `97` est l'équivalent ASCII de la lettre `a`. Quand vous stockez le caractère `a` dans une variable de type `char`, vous stockez en réalité la valeur `97`. La question que l'on peut maintenant se poser est : comment le programme sait-il si une variable de type `char` est un caractère ou un nombre ? L'utilisation le dira :

- Si une variable `char` est utilisée à un endroit du programme où un caractère est attendu, elle sera interprétée comme un caractère.
- Si une variable `char` est utilisée à un endroit du programme où un nombre est attendu, elle sera interprétée comme un nombre.

Les variables caractère

Comme toutes les autres variables, une variable `char` doit être déclarée , et elle peut être initialisée dans la même instruction :

```
char a, b, c;     /* Déclaration de trois variables char non */
                  /* initialisées */
char code = 'x';  /* Déclaration d'une variable char appelée code */
                  /* dans laquelle on stocke le caractère x */
code = '!';       /* On stocke le caractère ! dans la variable code */
```

Pour créer des constantes caractère, le caractère doit être entouré de guillemets simples. Le compilateur le convertit automatiquement dans le code ASCII correspondant et la valeur du code est attribuée à la variable.

Vous pouvez créer des constantes caractère symboliques en utilisant l'ordre `#define` ou le mot clé `const`.

```
#define IX 'x'
char code = IX;    /* code égal 'x' */
const char A = 'Z';
```

Le Listing 10.1 vous montre la nature numérique du stockage des caractères en utilisant la fonction `printf()`. Cette fonction permet d'afficher indifféremment des caractères ou des nombres. `%c` dans la chaîne format demande à `printf()` d'afficher un caractère, alors que `%d` demande un entier décimal. Le Listing 10.1 initialise deux variables de type `char` et les affiche en mode caractère, puis en mode numérique.

Listing 10.1 : Démonstration de la nature numérique des variables de type char

```
/* Démonstration de la nature numérique des variables char */
#include <stdio.h>
#include <stdlib.h>

/* Déclaration et initialisation de deux variables char */

char c1 = 'a';
char c2 = 90;

int main()
{
/*   Affichage de la variable c1 comme caractère, puis comme nombre   */

    printf("En mode caractère,la variable c1 est %c\n", c1);
    printf("En mode nombre, la variable c1 est %d\n", c1);

/* La même chose pour la variable c2 */

    printf("En mode caractère, la variable c2 est %c\n", c2);
    printf("En mode nombre, la variable c2 est %d\n", c2);

    exit(EXIT_SUCCESS);
}
```



```
En mode caractère, la variable c1 est a
En mode nombre, la variable c1 est 97
En mode caractère, la variable c2 est Z
En mode nombre, la variable c2 est 90
```

La valeur d'une variable de type `char` est comprise entre –128 et 127 (voir Tableau 3.2) alors que le code ASCII attribue les valeurs 0 à 255. En fait, ce code est divisé en deux. Le code ASCII standard est compris entre 0 et 127. Il permet de coder les lettres, les chiffres, la ponctuation et autres symboles du clavier. Le code ASCII étendu (128 à 255) représente tous les caractères spéciaux et symboles graphiques (liste en Annexe A).Vous pouvez donc utiliser des variables de type `char` pour du texte standard. Pour afficher des caractères ASCII étendus, il faudra déclarer des variables de type `unsigned char`.

Listing 10.2 : Affichage des caractères ASCII étendus

```
 1:  /* Affichage des caractères ASCII étendus */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  unsigned char x;    /* unsigned pour ASCII étendu */
 6:
 7:  int main()
 8:  {
 9:  /* Affichage des caractères ASCII étendus de 180 à 203 */
10:
11:   for (x = 180; x < 204; x++)
12:   {
13:    printf("Le code ASCII %d correspond au caractère %c\n", x, x);
14:   }
15:
16:    exit(EXIT_SUCCESS);
17:  }
```

```
Le code ASCII 180 correspond au caractère ┤
Le code ASCII 181 correspond au caractère ╡
Le code ASCII 182 correspond au caractère ╢
Le code ASCII 183 correspond au caractère ╖
Le code ASCII 184 correspond au caractère ╕
Le code ASCII 185 correspond au caractère ╣
Le code ASCII 186 correspond au caractère ║
Le code ASCII 187 correspond au caractère ╗
Le code ASCII 188 correspond au caractère ╝
Le code ASCII 189 correspond au caractère ╜
Le code ASCII 190 correspond au caractère ╛
Le code ASCII 191 correspond au caractère ┐
Le code ASCII 192 correspond au caractère └
Le code ASCII 193 correspond au caractère ┴
Le code ASCII 194 correspond au caractère ┬
Le code ASCII 195 correspond au caractère ├
Le code ASCII 196 correspond au caractère ─
Le code ASCII 197 correspond au caractère ┼
Le code ASCII 198 correspond au caractère ╞
Le code ASCII 199 correspond au caractère ╥
Le code ASCII 200 correspond au caractère ╙
Le code ASCII 201 correspond au caractère ╓
Le code ASCII 202 correspond au caractère ╩
Le code ASCII 203 correspond au caractère ╦
```

Analyse

La ligne 5 de ce programme déclare une variable de type `unsigned char`. Cela nous donne un intervalle de valeurs compris entre 0 et 255. Comme pour les autres types de données numériques, vous ne devez pas initialiser une variable `char` avec une valeur qui n'appartient pas à l'intervalle autorisé. La variable `x` est initialisée à 180 en ligne 11. Chaque exécution de l'instruction `for` incrémente la valeur de `x` de 1, jusqu'à la valeur 204. Cette boucle affiche chaque fois la valeur numérique de `x` et le caractère ASCII correspondant.

Conseils

À faire

Utiliser `%c` *pour afficher l'équivalent caractère d'un nombre.*

À ne pas faire

Utiliser les guillemets " " pour initialiser une variable caractère.

À faire

Utiliser les guillemets simples (apostrophes) pour initialiser une variable caractère.

À ne pas faire

Stocker la valeur d'un caractère ASCII étendu dans une variable `signed char`*.*

À faire

Étudier les caractères ASCII de l'Annexe A pour savoir ce que vous pouvez afficher.

Les chaînes

Les variables de type `char` ne peuvent recevoir qu'un caractère, leur utilisation est donc limitée. Il n'existe pas de type de donnée pour les chaînes de caractères. Un nom ou une adresse sont des exemples de chaînes de caractères. Le langage C manipule ce genre d'informations à l'aide des tableaux de caractères.

Tableaux de caractères

Pour stocker une chaîne de six caractères, il faut déclarer un tableau de type `char` avec sept éléments :

```
char chaine[7];
```

Une chaîne est une séquence de caractères qui se termine par le caractère nul `\0`. C'est le septième élément. Bien qu'il soit représenté par deux caractères (antislash et zéro), le caractère nul est interprété comme un seul caractère et sa valeur ASCII est 0.

Si un programme C stocke la chaîne `Alabama`, il stocke les sept caractères `A`, `l`, `a`, `b`, `a`, `m` et `a`, suivis du caractère nul. Il faut donc un tableau de huit éléments.

La taille d'une variable de type `char` est de un octet. Le nombre d'octets d'un tableau de caractères sera donc égal au nombre d'éléments.

Initialiser les tableaux de caractères

Les tableaux de caractères peuvent être initialisés dans l'instruction de déclaration de cette façon :

```
char chaine[10] = { 'A', 'l', 'a', 'b', 'a', 'm', 'a', '\0'};
```

Vous pouvez aussi utiliser la *chaîne littérale*, séquence de caractères entre guillemets, qui est plus facile à écrire et à lire :

```
char chaine[10] = "Alabama";
```

Dans ce cas, le compilateur ajoute automatiquement le caractère nul à la fin de la chaîne. De même, si la taille du tableau n'a pas été indiquée, le compilateur la calculera. La ligne suivante, par exemple, crée et initialise un tableau de huit éléments :

```
char chaine[] = "Alabama";
```

Le caractère nul permet aux fonctions qui manipulent des chaînes de caractères de connaître la longueur de la chaîne. Si vous l'avez oublié, la fonction n'a aucun autre moyen de déterminer la fin de la chaîne ; elle va continuer à traiter les données en mémoire tant qu'elle ne rencontre pas de caractère nul.

Chaînes et pointeurs

Une chaîne de caractères est stockée dans un tableau de type `char`, et la fin de cette chaîne est représentée par le caractère nul. Pour définir une chaîne, il suffit donc de pointer au début de cette chaîne. L'utilisation du nom du tableau dans lequel elle est stockée, non suivi de crochets, est la méthode standard d'accès à une chaîne de caractères.

La bibliothèque standard de C contient de nombreuses fonctions qui manipulent des chaînes de caractères. Pour passer une chaîne en argument, la fonction s'attend à recevoir le nom du tableau dans lequel elle est stockée. C'est la méthode à utiliser avec les fonctions de la bibliothèque, en particulier avec `printf()` et `puts()` que nous avons étudiées.

Les chaînes sans tableaux

Nous venons de voir que le nom du tableau qui contient la chaîne est un pointeur sur le début de la chaîne, et que le caractère nul représente la fin de cette chaîne. Le rôle du tableau ne consiste qu'à fournir de la place mémoire pour stocker la chaîne de caractères.

Pour se passer du tableau, il faut pouvoir s'allouer de la place mémoire, définir un pointeur en début de chaîne et placer le caractère nul à la fin. Il existe deux méthodes : dans la

première, pour une chaîne littérale dont la taille est définie au moment de la compilation du programme, la mémoire est allouée une bonne fois pour toutes au lancement du programme. La seconde consiste à utiliser la fonction `malloc()` qui alloue la mémoire au moment de l'exécution du programme ; le procédé s'appelle *allocation dynamique*.

Allouer la mémoire nécessaire à la compilation

Le début de la chaîne est représenté par un pointeur vers une variable `char`. Par exemple :

```
char *message;
```

Cette instruction déclare le pointeur `message` vers une variable de type `char`, mais le pointeur ne pointe encore sur rien. Si vous écrivez :

```
char *message = "Le fantôme \du grand César !";
```

vous obtenez le stockage de la chaîne de caractères `Le fantôme du grand César !` (avec un caractère nul à la fin) quelque part en mémoire, et le pointeur `message` est initialisé pour pointer sur le premier caractère. Il est inutile de connaître l'emplacement mémoire exact qui est géré par le système. Vous allez utiliser le pointeur pour accéder à la chaîne.

Voici une instruction équivalente à la précédente :

```
char message[] = "Le fantôme \du grand César !";
```

Cette méthode d'allocation de mémoire est parfaite quand vous connaissez vos besoins en écrivant le programme. Si le programme doit lire la chaîne de caractères à partir du clavier, par exemple, vous ne pourrez pas prévoir la taille de cette chaîne. Il faudra utiliser la fonction `malloc()` pour allouer la mémoire de façon dynamique.

La fonction *malloc()*

La fonction `malloc()` est une des fonctions de C qui permettent de réserver de l'espace mémoire. On lui transmet en argument le nombre d'octets nécessaires, elle se charge de trouver et de réserver un bloc de mémoire libre, puis renvoie l'adresse du premier octet de ce bloc au programme appelant.

La donnée renvoyée par la fonction `malloc()` est un pointeur de type `void`. Un pointeur de ce type sera compatible avec tous les types de données.

Syntaxe de la fonction *malloc()*

```
#include <stdlib.h>
void *malloc(size_t taille);
```

La fonction `malloc()` réserve un bloc de mémoire du nombre d'octets indiqué dans `taille`. Cette méthode d'allocation de mémoire permet d'optimiser la gestion de la mémoire de votre ordinateur. L'appel de cette fonction ne peut se faire que si vous avez inclus le fichier en-tête stdlib.h.

La valeur renvoyée par `malloc()` est un pointeur vers le bloc de mémoire qui a été réservé. Si la recherche de la mémoire a échoué, la valeur renvoyée est nulle. Il est donc nécessaire de contrôler cette valeur même si la taille de la mémoire demandée est petite.

Exemple 1

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    /* Allocation de mémoire pour une chaîne de 100 caractères */
    char *ch;
    if ((ch = malloc(100*sizeof(*ch))) == NULL)
    {
         printf("Il n'y a pas assez de mémoire \n");
         exit (EXIT_FAILURE);
    }
    printf("La mémoire est allouée !"\n );
    exit(EXIT_SUCCESS);
}
```

Exemple 2

```
/* Allocation de mémoire pour un tableau de 50 entiers */
int *nombres;
nombres = malloc(50 * sizeof(*nombres));
```

Exemple 3

```
/* Allocation de mémoire pour un tableau de 10 valeurs à virgule
   flottante */
float *nombres;
nombres = malloc(10 * sizeof(*nombres));
```

Utilisation de *malloc()*

`malloc()` peut servir à réserver de la mémoire pour un seul caractère. On déclare un pointeur vers une variable de type `char` :

```
char *ptr;
```

On appelle la fonction en lui passant la taille du bloc mémoire désiré. Une variable de type `char` a une taille d'un octet et la valeur renvoyée sera attribuée au pointeur :

```
ptr = malloc(1);
```

Cet octet réservé n'a pas de nom, `ptr` est donc le seul moyen d'y accéder. Pour y stocker le caractère `x`, vous devez écrire :

```
*ptr = 'x';
```

Pour réserver la mémoire nécessaire à une chaîne de caractères, la procédure est identique, mais il faut déterminer la taille maximale de cette chaîne. Notre exemple consiste à réserver de la mémoire pour une chaîne de 99 caractères, plus un pour le caractère nul. On commence en déclarant un pointeur vers une variable `char`, puis on appelle la fonction :

```
char *ptr;
ptr = malloc(100*sizeof(*ptr));
```

`ptr` pointe maintenant sur un bloc de 100 octets qui peut recevoir une chaîne de caractères.

Avec la fonction `malloc()`, la mémoire n'est réservée qu'au moment où on en a besoin. Bien sûr, la mémoire n'est pas infinie. La taille de la mémoire disponible dépend de la taille de la mémoire installée sur votre ordinateur et des besoins des autres programmes en cours d'exécution. Si la mémoire libre n'est pas suffisante, la fonction `malloc()` renvoie la valeur 0. Votre programme doit tester cette valeur pour vérifier que la mémoire demandée à bien été attribuée. Vous pouvez tester la valeur renvoyée par `malloc()` en la comparant à la constante symbolique `NULL` qui est définie avec stdlib.h.

Listing 10.3 : La fonction malloc()

```
/* Utilisation de la fonction malloc() pour réserver de la */
/* mémoire pour une chaîne. */

#include <stdio.h>
#include <stdlib.h>

char count, *ptr, *p;

int main()
{
/* Allocation d'un bloc de 35 octets. Test du résultat. */
/* La fonction de bibliothèque exit() termine le programme. */

   ptr = malloc(35 * sizeof(*ptr));

   if (ptr == NULL)
   {
       puts("Erreur d'allocation de la mémoire.");
```

Listing 10.3 : La fonction malloc() ***(suite)***

```
19:         exit(EXIT_FAILURE);
20:     }
21:
22:  /* On stocke dans la chaîne les valeurs 65 à 90, */
23:  /* qui sont les codes ASCII de A-Z. */
24:
25:  /* p est un pointeur qui permet de se déplacer dans la chaîne. */
26:  /* ptr pointe toujours sur le début de la chaîne. */
27:
28:
29:     p = ptr;
30:
31:     for (count = 65; count < 91 ; count++)
32:        *p++ = count;
33:
34:  /* On ajoute le caractère nul de fin. */
35:
36:     *p = '\0';
37:
38:  /* Affichage de la chaîne sur l'écran. */
39:
40:     puts(ptr);
41:
42:     exit(EXIT_SUCCESS);
43:  }
```



```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Analyse

Ce programme est un exemple d'utilisation simple de la fonction `malloc()`. Il contient de nombreuses lignes de commentaires qui expliquent son fonctionnement. La ligne 5 appelle le fichier en-tête stdlib.h pour la fonction `malloc()` et la ligne 4 appelle stdio.h pour la fonction `puts()`. La ligne 7 déclare la variable `char` et les deux pointeurs qui seront utilisés par le programme. Aucune de ces variables n'est encore initialisée.

La ligne 14 appelle la fonction `malloc()` en lui transmettant le paramètre 35 multiplié par la taille du type, à savoir dans notre cas celle de `char`. Il aurait été possible de mettre `sizeof(char)` au lieu de `sizeof(*ptr)` ligne 14, ce que l'on trouve d'ailleurs assez souvent. Cependant, multiplier par `sizeof(char)` est inutile car `sizeof(char)` vaut 1 par définition en C (attention, ce n'est pas le cas dans tous les langages, comme par exemple le Perl). Par contre, si l'envie vous venait de changer le type des éléments pointés par `ptr`, vous n'auriez pas à modifier la ligne 14 si vous pensez à multiplier par la taille de ce type comme nous l'avons écrit. L'opérateur `sizeof()` offre une méthode simple pour obtenir un code portable.

Ne supposez jamais que `malloc()` a trouvé la mémoire que vous lui demandiez. La ligne 16 vous montre comment contrôler facilement si votre demande a été satisfaite. Si la valeur de `ptr` est nulle, les lignes 18 et 19 vous envoient un message d'erreur et terminent le programme.

Le pointeur `p` est initialisé en ligne 29 avec la même adresse que le pointeur `ptr`. `p` est utilisé par la boucle `for` pour stocker les données dans le bloc mémoire réservé. La variable `count` de la ligne 31 est initialisée à 65 et incrémentée de 1 à chaque exécution de la boucle jusqu'à la valeur 91. Chaque valeur de `count` est stockée à l'adresse pointée par `p`. Ce pointeur est bien sûr incrémenté chaque fois que `count` change de valeur.

Les valeurs 65 à 91 correspondent, en code ASCII, aux 26 lettres majuscules de l'alphabet. La boucle `for` se termine quand l'alphabet complet a été stocké dans les emplacements mémoire pointés. La ligne 36 stocke le caractère nul à la dernière adresse pointée par `p`. Le pointeur `ptr` contient toujours l'adresse de la première valeur, `A`. Vous pouvez maintenant utiliser cet alphabet comme une chaîne de caractères ; les lettres seront traitées une par une jusqu'à la valeur nulle. La fonction `puts()` en ligne 40 vous affiche les valeurs qui ont été stockées.



À ne pas faire

Allouer plus de mémoire que nécessaire. Cette ressource n'est pas inépuisable, il faut l'utiliser avec parcimonie, même encore aujourd'hui où l'on compte la mémoire vive en gigaoctets.

Essayer de stocker une nouvelle chaîne de caractères dans un tableau déclaré et initialisé avec une chaîne de taille inférieure. Dans cette déclaration, par exemple :

char une_chaine[] = "NO";

`une chaine` *pointe sur "NO". Si vous essayez de stocker "YES" dans ce tableau, vous risquez d'avoir de sérieux problèmes. Ce tableau contenait trois caractères :* `N`*,* `O` *et le caractère nul. Si vous y stockez les quatre caractères* `Y`*,* `E`*,* `S` *et nul, vous ne savez pas quelle donnée sera écrasée par le caractère nul.*

Affichage de chaînes et de caractères

Un programme qui manipule des chaînes de caractères a souvent besoin de les afficher à l'écran. Les deux fonctions utilisées sont `puts()` et `printf()`.

La fonction *puts()*

La fonction `puts()` reçoit en argument le pointeur vers la chaîne de caractères qu'elle va afficher à l'écran. Cette fonction permet d'afficher les chaînes littérales aussi bien que les variables, et elle ajoute un retour à la ligne à la suite de chaque chaîne qui lui est confiée.

Listing 10.4 : La fonction puts()

```
 1:  /* Affichage de texte à l'écran avec puts(). */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  char *message1 = "C";
 6:  char *message2 = "est le";
 7:  char *message3 = "meilleur";
 8:  char *message4 = "langage de";
 9:  char *message5 = "programmation !";
10:
11:  int main()
12:  {
13:      puts(message1);
14:      puts(message2);
15:      puts(message3);
16:      puts(message4);
17:      puts(message5);
18:
19:      exit(EXIT_SUCCESS);
20:  }
```

```
C
est le
meilleur
langage de
programmation !
```

Analyse

`puts()` est une fonction standard de sortie pour laquelle il faut inclure le fichier en-tête stdio.h. Les lignes 5 à 9 de ce programme déclarent et initialisent cinq variables message différentes. Chacune de ces variables est un pointeur vers un caractère ou une variable chaîne. Les lignes 13 à 17 affichent tous les messages avec la fonction `puts()`.

La fonction *printf()*

Comme `puts()`, la fonction de bibliothèque `printf()` permet d'afficher des textes à l'écran. Elle utilise pour cela une chaîne format, qui contient des ordres de conversion et met en forme le texte à afficher. Quand ce texte contient une chaîne de caractères, l'ordre de conversion à utiliser est `%s`.

Quand `printf()` rencontre l'ordre de conversion %s dans sa chaîne format, elle l'associe à l'élément correspondant de sa liste d'arguments. Cet argument doit être un pointeur vers

le début de la chaîne que l'on veut afficher. printf() va afficher tous les caractères à partir de cette adresse, jusqu'à ce qu'elle rencontre la valeur 0. Par exemple :

```
char *str = "Un message à afficher";
printf("%s", str);
```

Il est possible d'afficher une combinaison de plusieurs chaînes de caractères avec du texte littéral et/ou des variables numériques.

```
char *banque = "Banque de France";
char *nom = "Jean Dupont";
int solde = 1000;
printf("Le solde à la %s de %s est de %d.", banque, nom, solde);
```

Le message affiché à l'écran sera :

```
Le solde à la Banque de France de Jean Dupont est de 1000.
```

Tous les détails concernant l'utilisation de cette fonction sont donnés dans le Chapitre 14.

Lecture des chaînes de caractères

Les deux fonctions de bibliothèque fgets() et scanf() complètent les deux précédentes en permettant la lecture de chaînes de caractères entrées au clavier. Avant de pouvoir lire une chaîne, un programme doit prévoir un endroit pour la stocker. Il peut suivre pour cela une des deux méthodes déjà étudiées : déclarer un tableau ou appeler la fonction malloc().

La fonction *fgets()*

La fonction fgets() lit une chaîne de caractères entrée au clavier par l'utilisateur. Quand cette fonction est appelée, elle lit tout ce qui est tapé sur le clavier, jusqu'à ce que l'utilisateur enfonce la touche de retour à la ligne (touche Entrée). La fonction prend en compte le retour à la ligne, ajoute un caractère nul en fin de chaîne et renvoie la chaîne (tronquée si nécessaire à la taille indiquée en argument) au programme appelant. Cette chaîne est stockée à l'adresse indiquée par le pointeur de type char qui a été passé à fgets(). Pour appeler fgets() dans votre programme, vous devez inclure le fichier en-tête stdio.h.

Listing 10.5 : Utilisation de fgets() pour lire une chaîne de données entrée au clavier

```
/* Exemple d'utilisation de la fonction de bibliothèque fgets().  */

#include <stdio.h>

```

Listing 10.5 : Utilisation de fgets() pour lire une chaîne de données entrée au clavier *(suite)*

```
 5:  /* Allocation d'un tableau de caractères pour recevoir les données. */
 6:
 7:  char input[81];
 8:
 9:  int main()
10:  {
11:     puts("Saisissez votre texte, puis appuyez sur Entrée");
12:     fgets(input,sizeof(input), stdin));
13:     printf("Vous avez tapé: '%s'\n", input);
14:
15:     exit(EXIT_SUCCESS);
16:  }
Saisissez votre texte, puis appuyez sur Entrée
Cela est un test
Vous avez tapé 'Cela est un test
'
```

Analyse

Le premier argument de la fonction `fgets()` est l'expression `input`. `input` est le nom d'un tableau de type `char` et donc un pointeur vers le premier élément. Le tableau est déclaré en ligne 7 avec 81 éléments, qui correspondent aux 80 colonnes de votre fenêtre (la ligne la plus longue possible) plus le caractère nul. Le second argument est la taille maximale acceptée. Nous utilisons pour cela la taille du tableau (`sizeof(input)`). Le troisième argument est un descripteur de flux (que nous retrouverons au chapitre 16). Le descripteur de l'entrée standard est `stdin`.

La fonction `fgets()` peut renvoyer une valeur au programme appelant. Cette valeur, qui a été ignorée dans le Listing 10.5, est un pointeur de type `char` qui correspond à l'adresse de stockage de la chaîne lue. C'est la même valeur qui a été transmise à la fonction, mais cela permet au programme de tester si une erreur est survenue.

Quand on utilise `fgets()`, ou toute autre fonction qui stocke des données à l'aide d'un pointeur, il faut être sûr que le pointeur est bien positionné sur un bloc de mémoire réservé. Il est facile de commettre l'erreur suivante :

```
char *ptr;
fgets(ptr, sizeof(*ptr)*n, stdin);
```

Le pointeur `ptr` a été déclaré sans être initialisé ; on ne peut pas savoir où il pointe. La fonction `fgets()` va donc stocker sa chaîne de caractère à l'adresse pointée par `ptr` en écrasant les données qui s'y trouvent. Vous devez être vigilant, car le compilateur ne détecte pas ce genre d'erreur et peut, au mieux, vous afficher un avertissement (WARNING).

Syntaxe de la fonction *fgets()*

```
#include <stdio.h>
char *fgets(char *str, int taille, FILE *flux);
```

Avec `stdin` en troisième argument, fonction `fgets()` lit une chaîne de caractères, `str`, à partir de l'entrée standard (le clavier). Une chaîne est constituée d'une séquence de caractères terminée par le caractère de retour à la ligne. Quand ce caractère est lu, le caractère nul est ajouté en fin de chaîne.

La fonction `fgets()` renvoie un pointeur vers la chaîne qu'elle vient de lire. En cas de problème, la valeur renvoyée est nulle.

Exemple

```
/* exemple fgets() */
#include <stdio.h>
#include <stdlib.h>
char line[256];
int main()
{
    printf("Entrez une chaîne de caractères :\n");
    fgets(line, sizeof(line), stdin);
    printf("\nVous avez tapé :\n");
    printf("%s\n",line);
    exit(EXIT_SUCCESS);
}
```



Il existe également une fonction gets() qui ne prend pour argument qu'un pointeur vers une chaîne de caractères. Cette fonction est à éviter car il n'y a aucun contrôle sur la longueur de cette chaîne et l'utilisateur risque de déborder, ce qui entraînerait des dysfonctionnements.

Lecture de chaînes avec la fonction *scanf()*

La fonction de bibliothèque `scanf()` permet de lire les données numériques entrées au clavier (Chapitre 7). Pour cela, cette fonction utilise une *chaîne format* qui lui indique comment les lire. Si vous introduisez l'ordre de conversion `%s`, `scanf()` pourra lire une chaîne de caractères. Comme pour la fonction `fgets()`, on passe à `scanf()` un pointeur vers le bloc mémoire qui contiendra la chaîne.

Pour cette fonction, le début de la chaîne est le premier caractère non blanc qui est lu. Il y a deux manières de lui indiquer la fin de la chaîne. Si `%s` est utilisé dans la chaîne format, la chaîne sera lue jusqu'au premier caractère blanc rencontré (celui-ci ne fera pas partie de la chaîne). Si la chaîne format contient `%ns` (ou *n* est une constante entière qui indique la longueur du champ), `scanf()` lira les *n* caractères, mais s'arrêtera au premier blanc

rencontré s'il arrive avant. Pour la même raison que vous devez utiliser `fgets()` et non `gets()`, vous indiquerez impérativement `%ns` dans la chaîne format et ne ferez jamais usage de `%s` avec `scanf()`. Sinon, cette fonction pourrait lire plus de caractères que la chaîne ne peut en contenir, impliquant ensuite un comportement inattendu du programme.

Vous pouvez lire plusieurs chaînes avec la même fonction `scanf()`, les règles précédentes s'appliqueront pour chaque chaîne. Par exemple :

```
scanf("%10s%14s%11s", s1, s2, s3);
```

Si vous tapez `janvier février mars` en réponse à cette instruction, `s1` sera égale à `janvier`, `s2` à `février` et `s3` à `mars`.

Si vous tapez `septembre` en réponse à l'instruction suivante :

```
scanf("%3s%3s%3s", s1, s2, s3);
```

les valeurs de `s1`, `s2` et `s3` seront respectivement `sep`, `tem` et `bre`.

Si vous ne tapez pas le nombre de chaînes requis, la fonction attendra les caractères suivants, et le programme ne continuera que lorsqu'elle les aura obtenus. Exemple :

```
scanf("%10s%14s%11s, s1, s2, s3);
```

Si la réponse à cette instruction est :

```
janvier février
```

le programme est suspendu et attend la troisième chaîne spécifiée dans la chaîne format. Si vous tapez plus de chaînes que n'en contient la chaîne format, les chaînes supplémentaires vont rester dans la mémoire tampon du clavier, en attendant la prochaine instruction `scanf()` ou une autre fonction d'entrées/sorties. Imaginons, par exemple, que vous répondiez `janvier février mars` aux instructions suivantes :

```
scanf("%10s%14s", s1, s2);
scanf("%11s", s3);
```

la fonction attribuera `janvier` à la chaîne `s1`, `février` à la chaîne `s2` et `mars` à `s3`.

La fonction `scanf()` renvoie une valeur entière égale au nombre d'éléments lus avec succès. On ignore cette valeur la plupart du temps. Si le programme ne lit que du texte, il faut choisir `fgets()` plutôt que `scanf()`. Cette dernière est particulièrement indiquée quand vous avez à lire une combinaison de textes et de valeurs numériques. Cela est illustré par le Listing 10.7. N'oubliez pas qu'il faut utiliser l'opérateur d'adresse (`&`) pour lire des variables numériques avec `scanf()`.

Listing 10.7 : Lecture de textes et de valeurs numériques avec scanf()

```
 1:  /* La fonction scanf(). */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  char lnom[81], fnom[81];
 6:  int count, id_num;
 7:
 8:  int main()
 9:  {
10:  /* message pour l'utilisateur. */
11:
12:     puts("Entrez vos nom, prénom et matricule séparés");
13:     puts("par un espace, puis appuyez sur Entrée.");
14:
15:  /* Lecture des trois chaînes. */
16:
17:     count = scanf("%80s%80s%d", lnom, fnom, &id_num);
18:
19:  /* Affichage des données. */
20:
21:     printf("%d chaînes ont été lues :  %s %s %d\n", count, fnom,
22:            lnom, id_num);
23:     exit(EXIT_SUCCESS);
24:  }
```



```
Entrez vos nom, prénom et matricule séparés par un espace,
puis appuyez sur Entrée.
Jones Bradley 12345
3 chaînes ont été lues : Bradley Jones 12345
```

Analyse

Nous avons vu que les paramètres de `scanf()` sont des adresses de variables. Dans le Listing 10.7, `lnom` et `fnom` sont des pointeurs (donc des adresses), mais `id num` est un nom de variable auquel il faut ajouter l'opérateur `&` avant de le transmettre à la fonction `scanf()` (ligne 17).

Résumé

Nous venons d'étudier les données de type `char` dans lesquelles on peut stocker un caractère. Chaque caractère est stocké sous forme de nombre fourni par le code ASCII. Ce type de variable peut aussi recevoir un nombre entier.

Une chaîne est une séquence de caractères terminée par un caractère nul. Les chaînes permettent de manipuler les données texte. En langage C, une chaîne de longueur *n* est stockée dans un tableau de type `char` de *n + 1* éléments.

Les fonctions qui gèrent la mémoire, comme `malloc()`, permettent à votre programme d'allouer dynamiquement de la mémoire. La fonction `malloc()` va réserver la taille mémoire nécessaire à votre programme. Sans ces fonctions, vous seriez obligé d'estimer la taille de cette mémoire et probablement d'en réserver plus que nécessaire.

Q & R

Q Quelle est la différence entre une chaîne et un tableau de caractères ?

R Une chaîne est une séquence de caractères terminée par le caractère nul. Un tableau est une séquence de caractères. Une chaîne est donc un tableau de caractères qui se terminerait par le caractère nul.

Si vous définissez un tableau de type char, le bloc mémoire alloué pour ce tableau sera de la taille exacte du tableau. Vous ne pourrez pas y stocker une chaîne plus longue.

Voici un exemple :

```
char dept[10] = "Côtes d'Armor";   /* Faux ! la chaîne est plus longue */
                                   /* que le tableau */
char dept2[10] = "22";             /* Correct, mais on réserve de la */
                                   /* mémoire pour rien */
```

Si vous définissez un pointeur vers une donnée de type char, ces restrictions ne s'appliquent pas. La seule variable à stocker est le pointeur, la chaîne se trouve quelque part en mémoire (vous devez savoir où). Il n'y a ni contrainte de longueur ni d'espace perdu, on peut faire pointer un pointeur sur une chaîne de n'importe quelle taille.

Q Pourquoi vaut-il mieux allouer la mémoire avec `malloc()` plutôt que de déclarer de grands tableaux pour y ranger ses données ?

R Déclarer de grands tableaux est la méthode la plus facile, mais vous ne faites pas le meilleur usage de votre mémoire. Quand vos programmes vont devenir de plus en plus gros, il sera très important de ne leur allouer de la mémoire que quand ils en auront besoin. Quand cette mémoire n'est plus nécessaire, on peut la libérer pour l'allouer à une autre variable ou tableau d'une autre partie du programme.

Q Les caractères ASCII étendus sont-ils disponibles sur tous les types de machines ?

R La plupart des ordinateurs supportent ces caractères, mais ce n'est pas le cas des plus anciens (dont le nombre diminue tous les jours).

Q Que se passe-t-il quand on stocke une chaîne de caractères dans un tableau trop petit ?

R Cela peut provoquer une erreur très difficile à localiser. Le compilateur ne la détectera pas et toutes les données en mémoire situées immédiatement après le tableau seront écrasées. Cela peut être un bloc mémoire non réservé, d'autres données, ou des informations vitales pour votre ordinateur. La gravité du résultat d'une telle manœuvre dépendra de l'importance des données perdues.

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Quelles sont les valeurs numériques correspondant aux caractères ASCII ?
2. Comment le compilateur C interprète-t-il un caractère entouré de guillemets simples ?
3. Quelle est la définition C d'une chaîne ?
4. Qu'est-ce qu'une chaîne littérale ?
5. Pourquoi faut-il déclarer un tableau de $n + 1$ éléments pour stocker une chaîne de longueur n ?
6. Comment le compilateur C interprète-t-il une chaîne littérale ?
7. En utilisant le tableau des caractères ASCII de l'Annexe A, trouvez les valeurs numériques correspondant aux caractères suivants :
   a) a.
   b) A.
   c) 9.
   d) un blanc.
   e) ╬.
   f) ♠
8. En utilisant le tableau des caractères ASCII de l'Annexe A, trouvez les caractères qui correspondent aux valeurs numériques suivantes :
   a) 73.
   b) 32.

c) 99.
d) 97.
e) 110.
f) 0.
g) 2.

9. Combien d'octets faudra-t-il allouer pour stocker les variables suivantes ? (En supposant qu'un caractère représente un octet.)
   a) `char *ch1 = { "chaîne 1" };`.
   b) `char ch2[] = { "chaîne 2" };`.
   c) `char chaîne3;`.
   d) `char ch4[20] = { "cela est la chaîne 4" };`.
   e) `char ch5[20];`.
10. Soit l'instruction :

```
char *string = "Une chaîne!";
```

   Déduisez-en la valeur des expressions :
   a) `string[0]`.
   b) `*string`.
   c) `string[11]`.
   d) `string[33]`.
   e) `*string+8`.
   f) `string`.

Exercices

1. Écrivez la déclaration de la variable `lettre` de type `char` en l'initialisant avec le caractère `$`.
2. Écrivez la ligne de code qui déclare un tableau de type `char` en l'initialisant avec la chaîne "les pointeurs sont fous !". La taille de ce tableau doit être juste suffisante pour y stocker la chaîne.
3. Écrivez l'instruction qui permet de réserver un bloc mémoire pour stocker la chaîne "les pointeurs sont fous !".
4. Écrivez le code qui permet de réserver un bloc mémoire pour stocker une chaîne de 80 caractères, lire cette chaîne au clavier et la stocker dans l'espace alloué.

5. Écrivez une fonction qui copie le contenu d'un tableau dans un autre (utilisez les exemples du Chapitre 9).
6. Écrivez une fonction qui lit deux chaînes de caractères. Comptez le nombre de caractères qui les composent et renvoyez un pointeur vers la chaîne la plus longue.
7. **TRAVAIL PERSONNEL** : Écrivez une fonction qui lit deux chaînes. Utilisez la fonction `malloc()` pour allouer la mémoire nécessaire au stockage des deux chaînes concaténées. Renvoyez le pointeur de la nouvelle chaîne.
8. **CHERCHEZ L'ERREUR :**

```
char une_chaine[10] = "cela est une chaîne";
```

9. **CHERCHEZ L'ERREUR :**

```
char *quote[100] = { "Souriez, Vendredi sera bientôt là !"};
```

10. **CHERCHEZ L'ERREUR :**

```
char *string1;
char *string2 = "second";
string1 = string2;
```

11. **CHERCHEZ L'ERREUR :**

```
char string1[];
char string2[] = "second";
string1 = string2;
```

12. **TRAVAIL PERSONNEL** : En utilisant le tableau de correspondance des caractères ASCII, écrivez un programme qui affiche une boîte à l'écran avec les caractères double-lignes.

11

Les structures

La réalisation de nombreuses tâches de programmation se trouve simplifiée par les *structures*. Une structure est une méthode de sauvegarde des données choisie par le programmeur ; elle répond donc exactement aux besoins du programme. Aujourd'hui, vous allez étudier :

- ce que représente une structure simple ou complexe ;
- la définition et la déclaration des structures ;
- le mode d'accès aux données des structures ;
- la création des structures pour stocker des tableaux et des tableaux des structures ;
- la déclaration de pointeurs dans les structures et de pointeurs vers de structures ;
- le passage de structures en arguments de fonctions ;
- la définition, la déclaration et l'utilisation des unions ;
- l'utilisation de définitions types avec les structures.

Les structures simples

Une structure contient une ou plusieurs variables groupées sous le même nom pour être traitées comme une seule entité. Contrairement aux variables stockées dans un tableau, les variables d'une structure peuvent être de types différents. Une structure peut contenir tous les types de données C, y compris les tableaux et les autres structures. Chaque variable d'une structure est appelée *membre* de cette structure. Le paragraphe qui suit vous en donne un exemple simple.

La définition et la déclaration des structures

Le code d'un programme graphique doit travailler avec les coordonnées des points de l'écran. Ces coordonnées sont représentées par une abscisse `x` pour la position horizontale, et une ordonnée `y` pour la position verticale. Vous pouvez définir une structure `coord` contenant les valeurs `x` et `y` d'un écran de cette façon :

```
struct coord {
    int x;
    int y;
};
```

Le mot clé `struct` identifie le début de la structure et informe le compilateur que `coord` est le type de structure. Les accolades renferment la liste des variables membres de la structure. Chaque membre doit être défini par un type de variable et un nom.

Notre exemple définit une structure `coord` qui contient deux variables entières `x` et y, mais ne déclare pas de structure. Il y a deux façons de déclarer les structures. La première consiste à faire suivre la définition d'une liste de noms de plusieurs variables :

```
struct coord {
    int x;
    int y;
} premier, second;
```

Ces instructions définissent la structure de type `coord` et déclarent deux structures appelées `premier` et `second` appartenant au type `coord`. `premier` contient deux membres entiers appelés `x` et `y`, tout comme `second`.

La seconde méthode consiste à déclarer les variables de la structure dans une autre partie du code du programme. Voici une autre syntaxe pour déclarer deux structures de type `coord` :

```
struct coord {
    int x;
    int y;
};
```

```
/* instructions ... */
struct coord premier, second;
```

L'accès aux membres d'une structure

Chaque membre d'une structure peut être utilisé comme une variable isolée du même type. Pour faire référence à un membre particulier, on sépare le nom de la structure concernée de celui du membre visé, avec l 'opérateur (`.`). Pour que la structure `premier` représente le point de l'écran de coordonnées `x=50`, `y=100`, on utilise la syntaxe suivante :

```
premier.x=50;
premier.y=100;
```

L'instruction suivante permet d'afficher les coordonnées d'écran stockées dans la structure `second :`

```
printf("%d,%d", second.x, second.y);
```

Le principal avantage des structures sur les variables est la possibilité de copier des informations entre structures de même type par une seule instruction simple. Dans le cas de notre exemple précédent, l'instruction :

```
premier = second;
```

est équivalente aux deux instructions suivantes :

```
premier.x = second.x;
premier.y = second.y;
```

Quand un programme utilise des structures complexes, possédant de nombreux membres, cette notation permet de gagner beaucoup de temps. Les autres avantages des structures apparaîtront chaque fois que des informations représentées par des types de variables différents devront être traités comme une seule entité. Dans une base de données destinée à un mailing, par exemple, chaque adresse sera une structure et les informations qui la constituent (nom, prénom, ville, etc.) en seront les membres.

Syntaxe du mot clé `struct`

```
struct nom_modèle {
    membre(s);
    /* instructions */
} occurrence;
```

Le mot clé `struct` permet de déclarer la structure. Une structure contient une ou plusieurs variables (`membre(s)`) groupées sous un même nom pour être traitées comme une seule

entité. Ces variables peuvent être de n'importe quel type : tableaux, pointeurs ou autres structures.

Le mot clé `struct` indique au compilateur le début de la définition d'une structure. Il est suivi du nom du modèle de la structure, puis des membres entre accolades. Il est possible de définir une ou plusieurs `occurrences` qui constituent les déclarations de structures. Une définition ne comportant pas de déclaration ne sert que de modèle pour les structures qui seront déclarées plus loin dans le programme. Une définition simple doit respecter la syntaxe suivante :

```
struct nom {
    membre(s);
    /* instructions ... */
};
```

Pour faire référence à ce modèle, la déclaration s'écrira :

```
struct nom occurrence;
```

Exemple 1

```
/* déclaration d'un modèle de structure appelé SSN */
struct SSN {
    int premier_trois;
    char dash1;
    int second_deux;
    char dash2;
    int dernier_quatre;
}
/* utilisation du modèle */
struct SSN client_ssn;
```

Exemple 2

```
/* Déclaration complète d'une structure */
struct date {
    char jour[2];
    char mois[2];
    char an[4];
} date_jour;
```

Exemple 3

```
/* Définition, déclaration et initialisation d'une structure */
struct heure {
    int heures;
    int minutes;
    int seconds;
} heure_naissance = { 8, 45, 0 };
```

Les structures plus complexes

Après avoir introduit les structures simples, nous pouvons étudier des types de structures plus intéressants. Ce sont les structures dont les membres sont d'autres structures ou des tableaux.

Structures contenant des structures

Les membres d'une structure peuvent être d'autres structures. Reprenons l'exemple précédent.

Supposons que votre programme ait besoin de traiter des rectangles. Un rectangle peut être représenté par les coordonnées de deux coins diagonalement opposés. Nous avons vu comment définir une structure pour enregistrer un point à l'aide de ses deux coordonnées. Pour définir le rectangle, nous avons besoin de deux structures telles que celle-ci. La structure `coord` ayant déjà été définie, vous pouvez définir la deuxième avec les instructions suivantes :

```
struct rectangle {
    struct coord hautgauche;
    struct coord basdroite;
};
```

La structure de type `rectangle` contient deux structures appartenant au type `coord` : `hautgauche` et `basdroite`.

Ces instructions ne constituent que le modèle de la structure, pour la déclarer, vous devez inclure une instruction du type :

```
struct rectangle maboite;
```

Vous auriez pu combiner définition et déclaration de cette façon :

```
struct rectangle {
    struct coord hautgauche;
    struct coord basdroite;
} maboite;
```

Pour accéder aux membres de type `int` de deux structures imbriquées, il faut utiliser deux fois l'opérateur (`.`) : `maboite.hautgauche.x`.

Cette expression fait référence au membre `x` du membre `hautgauche` de la structure `maboite` appartenant au type rectangle. L'exemple suivant est le code qui décrit un rectangle de coordonnées (0, 10), (100, 200) :

```
maboite.hautgauche.x = 0;
maboite.hautgauche.y = 10;
maboite.basdroite.x = 100;
maboite.basdroite.y = 200;
```

La Figure 11.1 vous explique les relations existant entre les différents membres et variables de cet exemple.

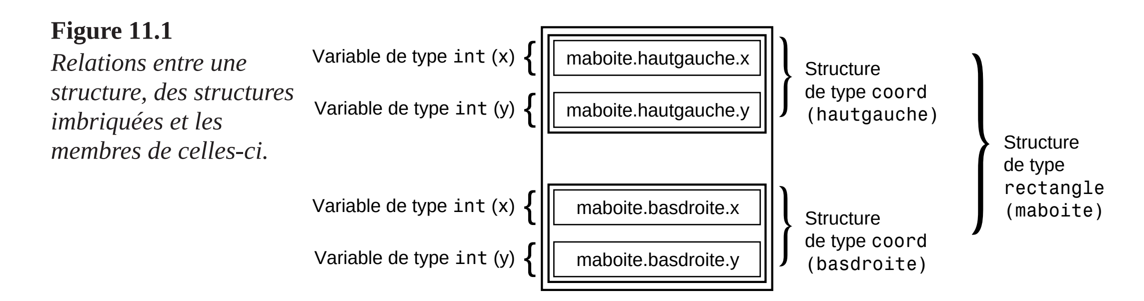


Figure 11.1
Relations entre une structure, des structures imbriquées et les membres de celles-ci.

Examinons le programme du Listing 11.1 qui nous présente un exemple de structures imbriquées. Ce programme demande à l'utilisateur les coordonnées d'un rectangle pour en calculer l'aire et l'afficher.

Listing 11.1 : Exemple de structure contenant une structure

```
/* Exemple de structures imbriquées. */

/* Ce programme reçoit les coordonnées des coins d'un rectangle
   et en calcule l'aire. On suppose que la coordonnée y du coin
   supérieur gauche est plus grande que la coordonnée y du coin
   inférieur droit, que la coordonnée x du coin inférieur droit
   est plus grande que la coordonnée x du coin supérieur gauche,
   et que toutes les coordonnées sont positives. */
#include <stdio.h>
#include <stdlib.h>

int longueur, largeur;
long aire;

struct coord{
  int x;
  int y;
};

struct rectangle{
  struct coord hautgauche;
  struct coord basdroite;
} maboite;

int main()
{
/* Lecture des coordonnées */

```

```
29:     printf("\nEntrez la coordonnée x du coin supérieur gauche : ");
30:     scanf("%d", &maboite.hautgauche.x);
31:
32:     printf("\nEntrez la coordonnée y du coin supérieur gauche : ");
33:     scanf("%d", &maboite.hautgauche.y);
34:
35:     printf("\nEntrez la coordonnée x du coin inférieur droit : ");
36:     scanf("%d", &maboite.basdroit.x);
37:
38:     printf("\nEntrez la coordonnée y du coin inférieur droit : ");
39:     scanf("%d", &maboite.basdroit.y);
40:
41:   /* Calcul de la longueur et de la largeur */
42:
43:     largeur = maboite.basdroit.x - maboite.hautgauche.x;
44:     longueur = maboite.basdroit.y - maboite.hautgauche.y;
45:
46:   /* Calcul et affichage de l'aire */
47:
48:     aire = largeur * longueur;
49:     printf("\nL'aire du rectangle est de %ld unités.\n", aire);
50:     exit(EXIT_SUCCESS);
51:   }
```

```
Entrez la coordonnée x du coin supérieur gauche : 1
Entrez la coordonnée y du coin supérieur gauche : 1
Entrez la coordonnée x du coin inférieur droit : 10
Entrez la coordonnée y du coin inférieur droit : 10
L'aire du rectangle est de 81 unités.
```

Analyse

La structure `coord` est définie aux lignes 15 à 18 de ce programme avec ses deux membres : `x` et `y`. Les lignes 20 à 23 déclarent et définissent la structure `maboite` appartenant au type `rectangle`. Les deux membres de la structure rectangle, `hautgauche` et `basdroit`, sont des structures de type `coord`.

Le programme demande les coordonnées du rectangle à l'utilisateur pour les stocker dans la structure `maboite` (lignes 29 à 39). Ces coordonnées sont au nombre de quatre, chaque structure `hautgauche` et `basdroit` ayant deux membres correspondant aux coordonnées `x` et `y`. Pour le calcul de l'aire, `x` et `y` étant dans une structure imbriquée, il faut associer le nom des deux structures pour accéder à leur valeur : `maboite.basdroit.x ou y`, et `maboite.hautgauche.x` ou `y`.

Le langage C n'impose aucune limite quant au nombre de structures que l'on peut imbriquer. Le bon sens et la mémoire disponible vous feront rarement dépasser trois niveaux d'imbrication en écrivant vos programmes.

Les tableaux membres de structures

Vous pouvez définir une structure constituée d'un ou de plusieurs tableaux. Les tableaux peuvent contenir tous les types de données C. Les instructions suivantes, par exemple, définissent un modèle de structure `data` qui contient un tableau d'entiers de 4 éléments appelé `x`, et un tableau de caractères de 10 éléments appelé `y` :

```
struct data{
    int x[4];
    char y[10];
};
```

Vous pouvez ensuite déclarer une structure `record` appartenant au type `data` avec l'instruction :

```
struct data record;
```

La Figure 11.2 représente cette structure de tableaux.

Vous pouvez remarquer que les éléments du tableau `x` occupent deux fois plus de place mémoire que les éléments du tableau `y`. En effet, une donnée de type `int` occupe deux octets en mémoire alors qu'une donnée de type `char` n'en occupe qu'un.

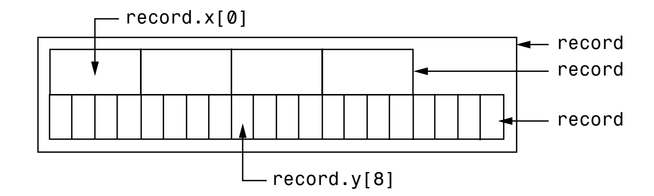


Figure 11.2
Organisation d'une structure contenant des tableaux.

Pour accéder aux éléments de tableaux qui sont membres d'une structure, on associe le nom du membre et l'index du tableau :

```
record.x[2] = 100;
record.y[1] = 'x';
```

Nous avons vu que les tableaux de caractères sont utilisés le plus souvent pour stocker des chaînes, et que le nom du tableau sans les crochets est un pointeur vers le premier élément du tableau. Nous pouvons en déduire que `record.y` est un pointeur vers le premier élément du tableau `y[]` de la structure `record`. Vous pourriez ainsi afficher le contenu de `y[]` avec la syntaxe suivante :

```
puts(record.y);
```

Étudions un autre exemple avec le Listing 11.2.

Listing 11.2 : Exemple d’une structure contenant des tableaux

```
 1:  /* Exemple d'utilisation d'une structure qui contient des */
 2:  /* tableaux. */
 3:  #include <stdio.h>
 4:  #include <stdlib.h>
 5:  /* Définition et déclaration d'une structure pour y ranger les */
 6:  /* données. Elle contient une variable de type float et deux */
 7:  /* tableaux de type char */
 8:  struct data{
 9:      float montant;
10:      char fnom[30];
11:      char pnom[30];
12:  } rec;
13:
14:  int main()
15:  {
16:  /* Lecture des données au clavier. */
17:
18:      printf("Entrez les nom et prénom du donateur,\n");
19:      printf("séparés par un espace :  ");
20:      scanf("%30s %30s", rec.fnom, rec.pnom);
21:
22:      printf("\nEntrez le montant du don :  ");
23:      scanf("%f", &rec.montant);
24:
25:  /* On affiche les informations. */
26:  /* Note: %.2f indique qu'une valeur à virgule flottante doit */
27:  /* être affichée avec deux chiffres après la virgule. */
28:
29:      printf("\nLe donateur %s %s a donné %.2f Euros\n",
30:              rec.fnom, rec.pnom, rec.montant);
30:      exit(EXIT_SUCCESS);
31:
32:  }
```

```
Entrez les nom et prénom du donateur,
séparés par un espace : Bradley Jones

Entrez le montant du don : 1000.00
Le donateur Bradley Jones a donné 1000.00 Euros
```

Analyse

Ce programme crée une structure qui contient les tableaux `fnom[30]` et `lnom[30]`. Ce sont deux tableaux de caractères qui vont contenir respectivement le nom et le prénom des personnes. La structure `data` est déclarée aux lignes 8 à 12. Ses membres sont les tableaux de caractères `fnom[30]` et `lnom[30]`, et la variable `montant`. Cette structure a été créée pour stocker le montant des dons des personnes qui seront enregistrées.

La structure `rec` de type `data` déclarée en ligne 12 est utilisée par le programme pour demander les valeurs à l’utilisateur (lignes 18 à 23) et pour les afficher (lignes 29 et 30).

Tableaux de structures

Le langage C est un langage très puissant et sa faculté de créer des tableaux de structures en est un bon témoignage. Voici pourquoi les tableaux de structures représentent un formidable outil de programmation.

La définition d'une structure doit "coller" au modèle des données avec lesquelles le programme doit travailler. En général, le programme a besoin de plusieurs modèles de données. Par exemple, dans un programme qui remplit la fonction de répertoire téléphonique, vous pouvez définir une structure contenant le nom et le numéro de téléphone de chaque personne :

```
struct entry{
    char fnom[10];
    char pnom[12];
};
```

Une liste telle que celle-ci doit pouvoir recevoir de nombreuses occurrences. La déclaration d'une structure de ce type ne sera donc pas très utile. Pour obtenir un vrai répertoire il faut créer un tableau de structures de type `entry` :

```
struct entry list[1000];
```

Cette instruction déclare un tableau `list` de 1000 éléments. Chacun de ces éléments est une structure de type `entry` qui sera identifiée par un index, comme tout élément de tableau. Chacune de ces structures regroupe trois éléments qui sont des tableaux de type `char`. La Figure 11.3 vous donne le diagramme de cette construction complexe.

Quand votre tableau de structures est déclaré, vous avez de nombreuses possibilités de manipulation des données. Par exemple, pour copier un élément de tableau dans un autre, vous pouvez écrire :

```
list[1] = list[5];
```

Cette instruction attribue à chaque membre de la structure `list[1]` la valeur contenue dans le membre correspondant de `list[5]`. Il est aussi possible de copier un membre de structure dans un autre :

```
strcpy(list[1].phone, list[5].phone);
```

Cette instruction copie la chaîne stockée dans `list[5].phone` dans `list[1].phone`. La fonction `strcpy()`, que vous étudierez au Chapitre 17, permet de copier une chaîne dans une autre chaîne. Une autre manipulation de donnée peut être la copie entre éléments de tableaux :

```
list[5].phone[1] = list[2].phone[3];
```

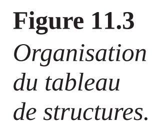
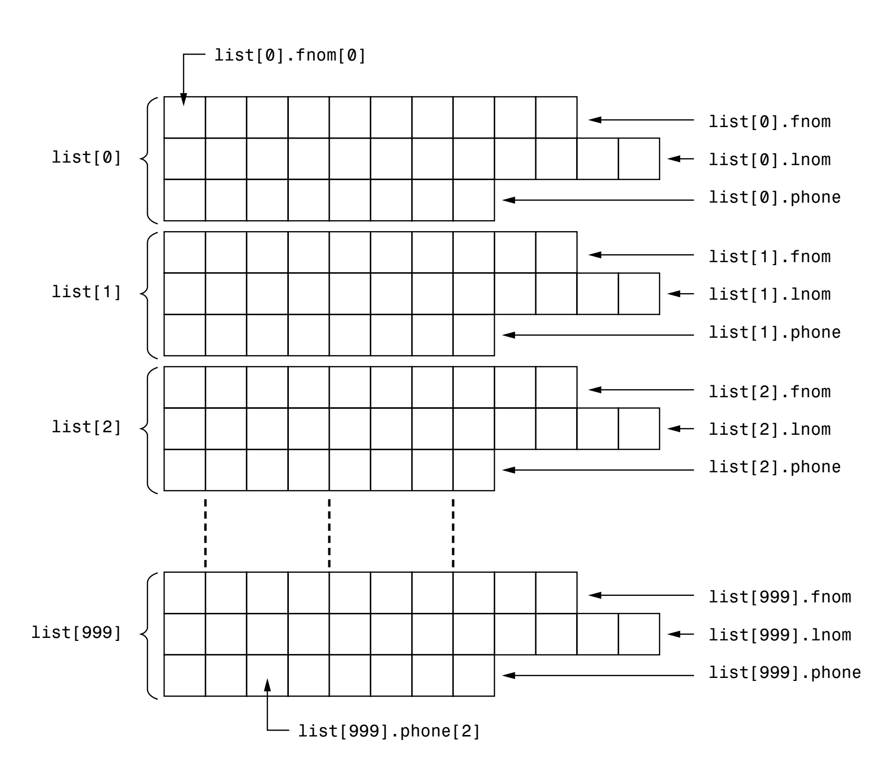


Figure 11.3
Organisation du tableau de structures.



Cette instruction copie le deuxième caractère du numéro de téléphone de `list[5]` en quatrième position du numéro de téléphone de `list[2]`.

Le Listing 11.3 présente un exemple de tableau de structures dont les membres sont des tableaux.

Listing 11.3 : Tableaux de structures

```
/* Exemple d'utilisation des tableaux de structures. */
#include <stdio.h>
#include <stdlib.h>

/* Définition d'une structure pour stocker les données. */

struct entry {
     char fnom[20];
     char pnom[20];
    char phone[10];
};

/* Déclaration d'un tableau de structures. */

struct entry list[4];
```

Listing 11.3 : Tableaux de structures *(suite)*

```

int i;

int main()
{

/* Boucle d'enregistrement de 4 personnes. */

     for (i = 0; i < 4; i++)
     {
     printf("\nEntrez le nom :  ");
     scanf("%20s", list[i].fnom);
     printf("Entrez le prénom : ");
     scanf("%20s", list[i].pnom);
     printf("Entrez le numéro de téléphone (xxxxxxxx) : ");
     scanf("%10s", list[i].phone);
     }

/* On saute deux lignes */

     printf("\n\n");

/* Affichage des données. */

     for (i = 0; i < 4; i++)
     {
     printf("Nom : %s %s", list[i].pnom, list[i].fnom);
     printf("\t\tPhone: %s\n", list[i].phone);
     }
     return 0;

}
```

```
Entrez le nom : Jones
Entrez le prénom : Bradley
Entrez le numéro de téléphone : 55591248

Entrez le nom : Aitken
Entrez le prénom : Peter
Entrez le numéro de téléphone : 52976543

Entrez le nom : Jones
Entrez le prénom : Melissa
Entrez le numéro de téléphone : 55983492

Entrez le nom : Dupont
Entrez le prénom : Charlotte
Entrez le numéro de téléphone : 35297651

Nom : Bradley Jones    Téléphone : 55591248
Nom : Peter Aitken     Téléphone : 52976543
Nom : Melissa Jones    Téléphone : 55983492
Nom : Charlotte Dupont Téléphone : 35297651
```

Analyse

Ce code source suit le même format que la plupart de ceux que nous avons déjà étudiés. Il commence par une ligne de commentaires et inclut le fichier stdio.h pour les entrées/sorties. Les lignes 7 à 11 définissent un modèle de structure appelé `entry` qui contient trois tableaux de caractères : `fnom[20]`, `lnom[20]`, et `phone[10]`. La ligne 15 utilise ce modèle pour déclarer le tableau `list` contenant quatre structures de type `entry`. Une variable de type `int` est définie en ligne 17 pour servir de compteur au programme. La fonction `main()` commence en ligne 19 et sa première fonction est d'exécuter quatre fois la boucle `for`. Le rôle de celle-ci est de récupérer les informations pour le tableau de structures (lignes 24 à 32). Vous pouvez remarquer que ce tableau utilise un index de la même façon que les tableaux étudiés au Chapitre 8.

Le programme marque une pause (ligne 36) en sautant deux lignes à l'écran avant l'affichage des données. Les valeurs contenues dans le tableau de structures sont obtenues en utilisant le nom du tableau indexé associé au nom du membre de la structure par l'opérateur (`.`).

Il est important de vous familiariser avec les techniques utilisées dans le Listing 11.3. De nombreuses tâches de programmation gagnent à être réalisées en utilisant des tableaux de structures dont les membres sont des tableaux.



À faire

Déclarer les structures en utilisant les mêmes règles de portée que pour les autres variables (voir Chapitre 12).

À ne pas faire

Oublier d'associer le nom de la structure à l'opérateur (`.`*) pour en utiliser un des membres.*

Confondre le nom de la structure et le nom du modèle de structure auquel elle appartient. Le modèle permet de définir un format, le nom de structure est une variable qui utilise ce format.

Oublier le mot clé `struct` *en déclarant une structure appartenant à un modèle prédéfini.*

Initialisation des structures

Comme tout autre type de variable C, les structures peuvent être initialisées quand elles sont déclarées. La procédure à suivre est la même que pour les tableaux. La déclaration est

suivie d’un signe égal puis, entre accolades, d’une liste de valeurs d’initialisation séparées par des virgules :

```
1:  struct vente {
2:      char client[20];
3:      char article [20];
4:      float montant;
5:  } mesventes = { "Acme Industries",
6:                  "ciseaux gauchers",
7:                  1000.00
8:      };
```

L’exécution de ces instructions donne les résultats suivants :

1. Définition d’un type de structure appelé `vente` (lignes 1 à 5).
2. Déclaration d’une structure appelée `mesventes` appartenant au type `vente` (ligne 5).
3. Initialisation du membre `mesventes.client` avec la chaîne `"Acme Industries"` (ligne 5).
4. Initialisation du membre `mesventes.article` avec la chaîne `"ciseaux gauchers"` (ligne 6).
5. Initialisation du membre `mesventes.montant` avec la valeur `1000.00` (ligne 7).

Dans le cas d’une structure dont les membres sont des structures, les valeurs d’initialisation doivent apparaître dans l’ordre. Elles seront stockées dans les membres en utilisant l’ordre dans lequel ces membres sont listés dans la définition de la structure.

```
 1:  struct client {
 2:      char societe[20];
 3:      char contact[25];
 4:  }
 5:
 6:  struct vente {
 7:      struct client acheteur;
 8:      char article[20];
 9:      float montant;
10:  } mesventes = { { "Acme Industries", "George Adams"},
11:     "ciseaux gauchers",
12:     1000.00
13:     };
```

L’exécution de ces instructions donne les résultats suivants :

1. Le membre `mesventes.acheteur.société` est initialisé avec la chaîne `"Acme Industries"` (ligne 10).
2. Le membre `mesventes.acheteur.contact` est initialisé avec la chaîne `"George Adams"` (ligne 10).

3. Le membre `mesventes.article` est initialisé avec la chaîne `"ciseaux gauchers"` (ligne 11).
4. Le membre `mesventes.montant` est initialisé à la valeur `1000.00` (ligne 12).

De la même façon, vous pouvez initialiser les tableaux de structures. Les données d'initialisation que vous listerez sont appliquées, dans l'ordre, aux structures du tableau. Pour déclarer, par exemple, un tableau de structures de type `vente` et initialiser les deux premiers éléments (donc, les deux premières structures), vous pouvez écrire :

```
 1:  struct client {
 2:      char société[20];
 3:      char contact[25];
 4:  };
 5:
 6:  struct vente {
 7:      struct client acheteur;
 8:      char article[20];
 9:      float montant;
10:  };
11:
12:
13:  struct vente y1990[100] = {
14:     { { "Acme Industries", "George Adams"},
15:     "ciseaux gauchers",
16:     1000.00
17:     }
18:     { { "Wilson & Co.", "Ed Wilson"},
19:     "peluche type 12",
20:     290.00
21:     }
22:  };
```

L'exécution de ce code donnera les résultats suivants :

1. Le membre `y1990[0].acheteur.société` est initialisé avec la chaîne `"Acme Industries"` (ligne 14).
2. Le membre `y1990[0].acheteur.contact` est initialisé avec la chaîne `"George Adams"` (ligne 14).
3. Le membre `y1990[0].article` est initialisé avec la chaîne `"ciseaux gauchers"` (ligne 15).
4. Le membre `y1990[0].montant` est initialisé avec la valeur `1000.00` (ligne 16).
5. Le membre `y1990[1].acheteur.société` est initialisé avec la chaîne `"Wilson & Co."` (ligne 18).
6. Le membre `y1990[1].acheteur.contact` est initialisé avec la chaîne `"Ed Wilson"` (ligne 18).

7. Le membre `y1990[1].article` est initialisé avec la chaîne `"Peluche type 12"` (ligne 19).
8. Le membre `y1990[1].montant` est initialisé avec la valeur `290.00` (ligne 20).

Structures et pointeurs

Les pointeurs étant un concept très important en langage C, il n'est pas surprenant de les retrouver avec les structures. Un membre de structure peut être un pointeur, et vous pouvez déclarer un pointeur vers une structure.

Les pointeurs membres d'une structure

Un pointeur qui est un membre d'une structure se déclare de la même façon qu'un pointeur qui ne l'est pas, en utilisant l'opérateur indirect (`*`) :

```
struct data {
    int *valeur;
    int *taux;
} premier;
```

Ces instructions définissent et déclarent une structure dont les deux membres sont des pointeurs vers des variables de type `int`. Cette déclaration doit être suivie de l'initialisation de ces pointeurs avec les adresses des variables pointées. Si nous supposons que `cout` et `interet` sont des variables de type `int`, l'initialisation des pointeurs suit la syntaxe suivante :

```
premier.valeur = &cout;
premier.taux = &interet;
```

Vous pouvez maintenant utiliser l'opérateur indirect (`*`). L'expression `*premier.valeur` a la valeur de la variable `cout` et l'expression `*premier.taux` a la valeur de la variable `interet`.

Les types de pointeurs les plus souvent utilisés comme membres de structures sont ceux qui pointent sur une chaîne de caractères. Les instructions suivantes déclarent un pointeur vers une variable `char` et l'initialise pour pointer sur une chaîne :

```
char *p_message;
p_message = "Le langage C";
```

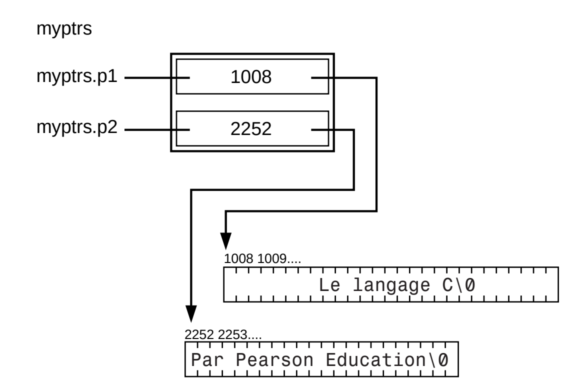
La même opération peut être réalisée si les pointeurs sont des membres de structure :

```
struct msg {
    char *p1;
```

```
    char *p2;
} myptrs;
myptrs.p1 = "Le langage C";
myptrs.p2 = "par Pearson Education";
```

La Figure 11.4 présente les résultats de l'exécution de ces instructions. Chaque pointeur membre de la structure pointe sur le premier octet d'une chaîne stockée quelque part en mémoire. Comparez ce schéma à celui de la Figure 11.3 qui montrait des données stockées dans une structure contenant des tableaux de type `char`.

Figure 11.4
Structure contenant des pointeurs vers des variables de type char.



L'utilisation de ce type de pointeur n'est pas différente de celle des pointeurs hors structure. Pour afficher les chaînes de notre exemple, vous pouvez écrire :

```
printf("%s %s", mesptrs.p1, mesptrs.p2);
```

Quelle différence y a-t-il entre un membre de structure qui est un tableau de type `char`, et un autre membre qui est un pointeur vers une variable de type `char` ? Ce sont deux méthodes de stockage des chaînes de caractères dans une structure. La structure suivante utilise les deux méthodes :

```
struct msg {
    char p1[30];
    char *p2;
} myptrs;
```

Le nom du tableau sans crochets étant un pointeur vers le premier élément du tableau, vous pouvez utiliser les deux membres de la structure de façon similaire :

```
strcpy(myptrs.p1, "Le langage C");
strcpy(myptrs.p2, "par Pearson Education");
/* instructions ... */
puts(myptrs.p1);
puts(myptrs.p2);
```

Si vous définissez un modèle de structure contenant un tableau de type `char`, chaque structure déclarée sur ce modèle occupera l'espace mémoire correspondant à la taille du tableau. En voici un exemple :

```
struct msg {
    char p1[10];
    char p2[10];
} myptrs;
...
strcpy(p1, "Montpellier"); /* incorrect, la chaîne est plus longue */
                           /* que le tableau                     */
strcpy(p2, "34");          /* correct, mais la chaîne étant plus   */
                           /* courte que le tableau, une partie de */
                           /* l'espace réservé restera inoccupé    */
```

Si vous utilisez l'autre méthode en définissant une structure contenant des pointeurs vers des variables de type `char`, cette contrainte de mémoire n'existe pas. Chaque structure déclarée n'occupera que l'espace mémoire nécessaire au pointeur. Les chaînes de caractères sont stockées dans une autre partie de la mémoire, indépendante de la structure. Le pointeur pourra pointer sur une chaîne de n'importe quelle longueur, celle-ci deviendra une partie de la structure tout en étant stockée en dehors.

Les pointeurs vers des structures

Un programme C peut déclarer et utiliser des pointeurs vers des structures exactement comme il peut déclarer des pointeurs vers tout autre type de donnée. Ces pointeurs sont souvent utilisés pour passer une structure comme argument à une fonction.

Voici comment un programme peut créer et utiliser des pointeurs vers des structures. La première étape est la définition de la structure :

```
struct part {
    int nombre;
    char nom[10];
};
```

La seconde est la déclaration d'un pointeur vers une variable de type `part`.

```
struct part *p_part;
```

Le pointeur ne peut être initialisé, car il n'existe pas encore de structure appartenant au type `part`. Il ne faut pas oublier que la définition du modèle de structure ne réserve pas de mémoire, c'est la déclaration d'une structure sur ce modèle qui le fait. La valeur d'un pointeur étant une adresse en mémoire, il faut déclarer une structure de type `part` avant de pouvoir pointer dessus :

```
struct part gizmo;
```

Cette instruction permet d'initialiser le pointeur :

```
p_part = &gizmo;
```

La valeur du pointeur p part représente l'adresse de la structure gizmo, tandis que *p part fait directement référence à gizmo.

Pour accéder aux membres de la structure gizmo, on utilise l'opérateur (.) de cette façon :

```
(*p_part).nombre = 100;
```

Les parenthèses sont nécessaires, car l'opérateur (.) est prioritaire sur l'opérateur (*). Cette instruction a attribué la valeur 100 au membre gizmo.nombre.

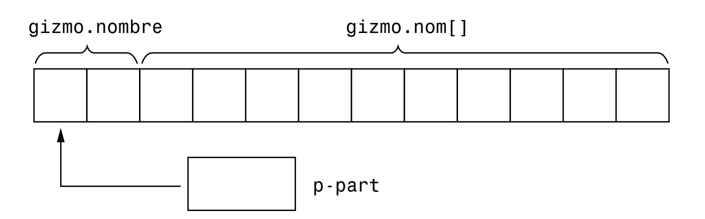


Figure 11.5
Un pointeur vers une structure pointe sur le premier octet de cette structure.

Il existe une autre technique pour accéder aux membres d'une structure avec le pointeur vers cette structure. Cette technique utilise *l'opérateur d'indirection*, représenté par le signe moins suivi du signe "supérieur à" (–>). Cet opérateur est placé entre le nom du pointeur et le nom du membre. Pour accéder au membre nombre de gizmo avec le pointeur p part, utilisez la syntaxe suivante :

```
p_part -> nombre
```

Étudions un autre exemple. Soit str une structure, p str un pointeur vers cette structure et memb un membre de str. Vous pouvez atteindre str.memb avec l'instruction suivante :

```
p_str -> memb
```

Il existe ainsi trois méthodes pour accéder au membre d'une structure :

- En utilisant le nom de la structure.
- Avec un pointeur vers cette structure et l'opérateur indirect (*).
- Avec un pointeur vers cette structure et l'opérateur d'indirection (–>).

Dans notre exemple, les trois expressions suivantes sont équivalentes :

```
str.memb
(*p_str).memb
p_str->memb
```

Pointeurs et tableaux de structures

Les tableaux de structures et les pointeurs de structures sont des outils de programmation très puissants. On peut les combiner en utilisant les pointeurs pour accéder aux structures qui sont des éléments de tableaux.

Reprenons, pour notre démonstration, cette définition de structure :

```
struct part {
    int nombre;
    char nom[10];
};
```

La structure étant définie, nous pouvons déclarer un tableau appartenant au type `part` :

```
struct part data[100];
```

Nous pouvons ensuite déclarer un pointeur vers une structure de type `part`, et l'initialiser pour pointer sur la première structure du tableau `data` :

```
struct part *p_part;
p_part = &data[0];
```

Nous aurions pu écrire la deuxième instruction de cette façon :

```
p_part = data;
```

Nous obtenons un tableau de structures de type `part` et un pointeur vers le premier élément du tableau. Un programme pourra afficher le contenu de ce premier élément avec l'instruction :

```
printf("%d %s", p_part -> nombre, p_part -> nom);
```

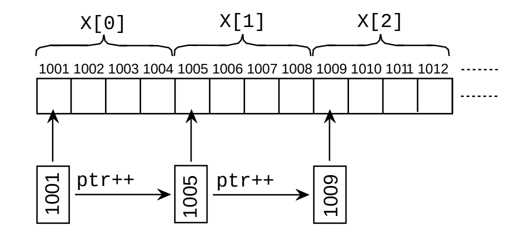
Pour afficher tous les éléments du tableau, il faudrait utiliser une boucle `for` et les afficher un par un à chaque exécution de la boucle. Pour accéder aux membres avec la notation pointeur, il faudra changer la valeur de `p part` pour qu'à chaque exécution de la boucle, l'adresse pointée soit celle de l'élément suivant (donc de la structure suivante). Voici comment réaliser cette opération.

Nous allons utiliser les pointeurs arithmétiques. L'opérateur unaire (++) a une signification particulière quand il est utilisé avec un pointeur. Il incrémente la valeur du pointeur, de la taille de l'objet pointé.

Les éléments de tableau étant stockés en mémoire de façon séquentielle, ces pointeurs arithmétiques sont particulièrement appropriés. Si, dans un programme, le pointeur pointe sur l'élément `n` d'un tableau, l'emploi de l'opérateur (++) sur ce pointeur le fera pointer sur l'élément `n+1`. La Figure 11.6 nous montre un tableau nommé `x[]` qui contient des éléments

ayant une taille de quatre octets (une structure contenant deux membres de type `int` par exemple). Le pointeur `ptr` a été initialisé pour pointer sur x[0]. Chaque fois que l'opérateur (++) sera utilisé, `ptr` pointera sur l'élément suivant.

Figure 11.6
L'opérateur ++ positionne le pointeur sur l'élément suivant.



Cela signifie qu'un programme peut accéder aux éléments d'un tableau de structures en incrémentant un pointeur. Cette notation est plus facile et plus concise que celle qui utilise l'index.

Listing 11.4 : Exemple d'accès aux éléments successifs d'un tableau en modifiant un pointeur avec l'opérateur (++)

```
/* Exemple de déplacement dans un tableau de structures */
/* en utilisant un pointeur. */
#include <stdio.h>
#include <stdlib.h>

#define MAX 4

/* Définition d'une structure, déclaration et initialisation */
/* d'un tableau de 4 structures. */

struct part {
    int nombre;
    char nom[10];
} data[MAX] = {1, "Smith",
    2, "Jones",
    3, "Adams",
    4, "Wilson"
    };

/* Déclaration d'un pointeur de structure de type part, */
/* et d'une variable compteur. */
struct part *p_part;
int count;

```

Listing 11.4 : Exemple d'accès aux éléments successifs d'un tableau en modifiant un pointeur avec l'opérateur (++) ***(suite)***

```
25:  int main()
26:  {
27:  /* Initialisation du pointeur sur le premier élément du tableau.*/
28:
29:      p_part = data;
30:
31:  /* Boucle qui permet de se déplacer dans le tableau */
32:  /* en incrémentant le compteur à chaque itération. */
33:
34:      for (count = 0; count < MAX; count++)
35:      {
36:          printf("A l'adresse %d :  %d %s\n", p_part, p_part->nombre,
37:          p_part->nom);
38:          p_part++;
39:      }
40:      exit(EXIT_SUCCESS);
41:
42:  }
```

```
A l'adresse 96 : 1 Smith
A l'adresse 108 : 2 Jones
A l'adresse 120 : 3 Adams
A l'adresse 132 : 4 Wilson
```

Analyse

Ce programme commence par déclarer et initialiser le tableau de structures `data` (lignes 11 à 18). Il définit ensuite le pointeur `p part` qui permettra de pointer sur la structure `data (ligne22)`. La première action de la fonction `main() à la ligne 29` est de faire pointer `p part` sur la structure de type `part` qui a été déclarée. La boucle `for` permet d'afficher tous les éléments en déplaçant le pointeur sur l'élément suivant à chacune de ses itérations de la ligne 34 à 39). Le programme donne aussi l'adresse de chaque élément.

Examinez les adresses affichées. Leur valeur sera différente sur votre système, mais la différence entre deux adresses sera la même, c'est-à-dire la taille de la structure `part` (12). Cela démontre bien que l'opérateur `(++)` augmente la valeur du pointeur de la taille de l'objet pointé.

Le passage de structures comme arguments de fonctions

Le Listing 11.5 vous montre comment passer une structure à une fonction. Ce programme a été obtenu en modifiant le Listing 11.2 : on utilise une fonction pour afficher les données à l'écran en remplacement des instructions de la fonction `main()` qui exécutaient cette tâche.

Listing 11.5 : Transmission d'une structure à une fonction

```
/* Transmission d'une structure à une fonction. */
#include <stdio.h>
#include <stdlib.h>

/* Déclaration et définition d'une structure stockant */
/* les données. */
struct data{
     float montant;
     char fnom[30];
    char lnom[30];
} rec;

/* Prototype de la fonction. Cette fonction ne renvoie pas de */
/* valeur, et son argument est une structure de type data. */

void print_rec(struct data x);

int main()
{
/* Lecture des données au clavier. */

    printf("Entrez les nom et prénom du donateur,\n");
    printf("séparés par un blanc :  ");
    scanf("%30s %30s", rec.fnom, rec.lnom);

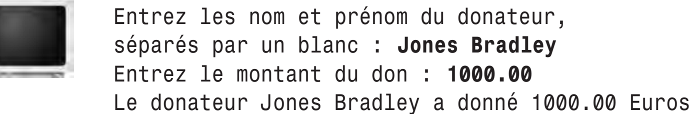
    printf("\nEntrez le montant du don :  ");
    scanf("%f", &rec.montant);

/* Appel de la fonction. */

    print_rec(rec);
    exit(EXIT_SUCCESS);

}

void print_rec(struct data x)
{
    printf("\nLe donateur %s %s a donné %.2f Euros\n", x.fnom, x.lnom,
    x.montant);
}
```



```
Entrez les nom et prénom du donateur,
séparés par un blanc : Jones Bradley
Entrez le montant du don : 1000.00
Le donateur Jones Bradley a donné 1000.00 Euros
```

Analyse

La ligne 16 de ce programme contient le prototype de la fonction qui va recevoir la structure. L'argument approprié dans notre cas est une structure de type `data`. Cet argument est repris à la ligne 36, dans l'en-tête de la fonction. Le passage de la structure se fait en ligne 31, en indiquant son nom dans la liste des arguments de l'appel de la fonction. Transmettre une structure à une fonction est aussi simple que de passer une variable.

La structure aurait pu être transmise en utilisant son adresse (un pointeur sur cette structure). N'oubliez pas, dans ce cas, l'opérateur (–>) pour atteindre les membres de la structure dans la fonction.



À ne pas faire

Confondre les tableaux et les structures !

À faire

Utiliser les pointeurs de structures, surtout avec les tableaux de structures.

À ne pas faire

Oublier que lorsque l'on incrémente un pointeur, il se déplace d'une distance équivalente à la taille des données pointées. Dans le cas d'un pointeur de structure, il s'agit de la taille de la structure.

À faire

Utiliser l'opérateur (–>) si vous travaillez avec un pointeur de structure.

Les unions

Unions et structures sont similaires. Une union est déclarée et utilisée comme une structure, mais on ne peut travailler qu'avec un seul de ses membres à la fois. La raison en est simple, tous les membres d'une union sont stockés un par un dans le même emplacement mémoire.

Définition, déclaration et initialisation des unions

Les unions sont définies et déclarées de la même façon que les structures, avec un mot clé différent. L'instruction suivante définit une union simple d'une variable `char` et d'une variable `integer` :

```
union partage {
    char c;
```

```
    int i;
};
```

Ce modèle d'union, appelé `partage`, va permettre de créer des unions qui pourront contenir soit un caractère `c`, soit un entier `i`. Contrairement à une structure qui aurait stocké les deux valeurs, l'union ne peut en recevoir qu'une à la fois. La Figure 11.7 vous montre comment apparaît l'union `partage` en mémoire.

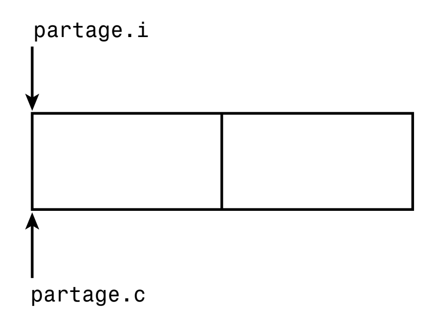


Figure 11.7
Une union ne peut recevoir qu'une valeur à la fois.

L'union peut être initialisée par son instruction de déclaration. Un seul membre pouvant être utilisé à la fois, pour éviter les erreurs, seul le premier peut être initialisé. Voici un exemple de déclaration et d 'initialisation d'une union de type `partage` :

```
union partage variable_generic = {'@'};
```

Accéder aux membres d'une union

On utilise les membres d'une union de la même façon que les membres de structure : avec l'opérateur (`.`). Il y a cependant une différence importante pour accéder aux membres d'une union. En effet, celle-ci sauvegarde ses membres à la suite les uns des autres, il est donc important d'y accéder un par un. Examinons l'exemple donné dans le Listing 11.6.

Listing 11.6 : Exemple de mauvaise utilisation d'une union

```
/* Exemple d'utilisation de plus d'un membre d'une union à la fois */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    union shared_tag {
    char      c;
    int       i;
    long      l;
   float     f;
   double    d;
   } shared;
```

Listing 11.6 : Exemple de mauvaise utilisation d’une union *(suite)*

```
13:
14:      shared.c = '$';
15:
16:      printf("\nchar c   = %c",  shared.c);
17:      printf("\nint i    = %d",  shared.i);
18:      printf("\nlong l   = %ld", shared.l);
19:      printf("\nfloat f  = %f",  shared.f);
20:      printf("\ndouble d = %f",  shared.d);
21:
22:      shared.d = 123456789.8765;
23:
24:      printf("\n\nchar c   = %c",  shared.c);
25:      printf("\nint i    = %d",  shared.i);
26:      printf("\nlong l   = %ld", shared.l);
27:      printf("\nfloat f  = %f",  shared.f);
28:      printf("\ndouble d = %f\n",  shared.d);
29:      exit(EXIT_SUCCESS);
30:
31:  }
```

Voici le résultat de l’exécution de ce programme :

```
char c    = $
int i     = 134513700
long l    = 134513700
float f   = 0.000000
double d  = 0.000000

char c    = 7
int i     = 1468107063
long l    = 1468107063
float f   = 284852666499072.000000
double d  = 123456789.876500
```

Analyse

Ce programme définit et déclare une union appelée `shared` aux lignes 6 à 12. `shared` contient cinq membres de types différents qui sont initialisés aux lignes 14 et 22. Les lignes 16 à 20, puis 24 à 28, de ce programme, présentent la valeur de chaque membre avec la fonction `printf()`.

À l’exception de `char c = $` et `double d = 123456789.876500`, les résultats obtenus sur votre machine pourront être différents. La variable `c` étant initialisée en ligne 14, c’est la seule valeur utilisable de l’union tant qu’un autre membre n’est pas initialisé à son tour. Le résultat de l’affichage des autres membres (`i`, `l`, `f` et `d`) est imprévisible.

En ligne 22, une valeur est stockée dans la variable `double d`. Vous pouvez remarquer que seul le résultat de l’affichage de `d` est correct. La valeur de `c` précédente a été écrasée par

celle de `d`. Cela met bien en évidence le fait qu'un seul emplacement mémoire est utilisé pour tous les membres de l'union.

Syntaxe du mot clé *union*

```
union tag {
membre(s);
/* instructions ... */
} occurrence;
```

Le mot clé `union` annonce la déclaration d'une union. Une union regroupe une ou plusieurs variables (`membre(s)`) sous un nom identique. Tous les membres de cette union occuperont le même emplacement mémoire.

Le mot clé `union` identifie le début de la définition, et il doit être suivi du nom `tag` donné au modèle de cette union. Ce nom est suivi de la liste des membres entre accolades. Avec `occurrence`, il est possible de déclarer une union appartenant au modèle défini. Si cette instruction ne contient pas de déclaration, c'est un simple modèle qui sera utilisé dans une autre partie du programme pour déclarer des unions.

Un modèle simple a le format suivant :

```
union tag {
    membre(s);
    /* instructions ... */
};
```

Voici la syntaxe d'une déclaration utilisant ce modèle :

```
union tag occurrence;
```

Exemple 1

```
/* Déclaration d'un modèle d'union appelé tag */
union tag {
    int nbr;
    char caractère;
}
/* utilisation du modèle */
union tag variable;
```

Exemple 2

```
/* Déclaration du modèle et d'une occurrence de l'union */
union type_generic {
    char c;
    int i;
    float f;
    double d;
} generic;
```

Exemple 3

```
/* Initialisation d'une union */
union date_mod {
    char date_complète[9];
    struct partie_date_mod {
      char mois[2];
      char séparateur1;
      char jour[2];
      char séparateur2;
      char année[2];
} partie_date;
} date = {"01/04/08"};
```

Le Listing 11.7 vous montre une utilisation pratique d'une union. Cet exemple est très simple, mais il représente l'usage le plus courant que l'on fait des unions.

Listing 11.7 : Utilisation pratique d'une union

```
/* Exemple typique d'utilisation d'une union */
#include <stdio.h>
#include <stdlib.h>

#define CARACTERE 'C'
#define INTEGER 'I'
#define FLOAT 'F'

struct generic_tag {
   char type;
   union shared_tag {
     char c;
     int i;
     float f;
   } shared;
};

void print_fonction(struct generic_tag generic);

int main()
{
   struct generic_tag var;

   var.type = CARACTERE;
   var.shared.c = '$';
   print_fonction(var);

   var.type = FLOAT;
   var.shared.f = (float) 12345.67890;
   print_fonction(var);

   var.type = 'x';
   var.shared.i = 111;
```

```
34:     print_fonction(var);
35:     exit(EXIT_SUCCESS);
36:  }
37:  void print_fonction(struct generic_tag generic)
38:  {
39:     printf("La valeur générique est...");
40:     switch(generic.type)
41:     {
42:       case CARACTERE :  printf("%c", generic.shared.c);
43:       break;
44:       case INTEGER :  printf("%d", generic.shared.i);
45:       break;
46:       case FLOAT :  printf("%f", generic.shared.f);
47:       break;
48:       default :  printf("de type inconnu : %c\n", generic.type);
49:       break;
50:     }
51:  }
```



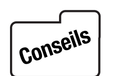
```
La valeur générique est...$
La valeur générique est...12345.678711
La valeur générique est...de type inconnu : x
```

Analyse

Ce programme donne un exemple simplissime de ce que l'on peut faire avec une union. Il permet de stocker plusieurs données de types différents dans le même emplacement mémoire. Le rôle de la structure `generic tag` est de ranger un caractère, un entier, ou un nombre avec une virgule flottante, dans la même zone. Cette zone est représentée par l'union `shared` qui a le même principe de fonctionnement que celle du Listing 11.6. Vous pouvez remarquer que la structure `generic tag` possède un champ supplémentaire appelé `type`. Ce champ est utilisé par le programme pour stocker le type de la variable contenue dans `shared`. Il permet d'éviter un mauvais usage de `shared` qui donnerait des valeurs erronées comme dans l'exemple du Listing 11.6.

Les noms des trois constantes `CARACTERE`, `INTEGER` et `FLOAT`, définies aux lignes 5, 6 et 7, ont été choisis pour faciliter la compréhension du code source. Les lignes 9 à 16 contiennent la définition de la structure `generic tag`, et la ligne 18 présente le prototype de la fonction `print fonction`. La structure `var` est déclarée ligne 22 puis initialisée pour recevoir un caractère en lignes 24 et 25. Cette valeur est affichée avec l'appel de la fonction `print fonction` en ligne 26. Les lignes 28 à 30, et 32 à 34 répètent ce processus avec les autres valeurs.

La fonction `print fonction` est le centre du programme. Elle permet d'afficher la valeur d'une variable de `generic tag` après avoir testé le type de cette variable pour éviter de faire la même erreur que dans le Listing 11.6.



À ne pas faire

Essayer d'initialiser plusieurs membres d'union en même temps.

À faire

Savoir quel membre de l'union est en cours d'utilisation. Si vous définissez un membre d'un certain type, et que vous essayez d'utiliser un autre type, le résultat est imprévisible.

À ne pas faire

Oublier que la taille d'une union est égale à celle du membre le plus grand.

Structures et typedef

Vous pouvez utiliser le mot clé `typedef` pour créer le synonyme d'une structure ou d'une union. Les instructions suivantes, par exemple, définissent `coord` comme synonyme de la structure désignée :

```
typedef struct {
    int x;
    int y;
} coord;
```

Vous pourrez ensuite déclarer des structures de ce type en utilisant `coord` :

```
coord hautgauche, basdroit;
```

Attention, le rôle de `typedef` est différent de celui du nom d'un modèle. Si vous écrivez :

```
struct coord {
    int x;
    int y;
};
```

`coord` est le nom du modèle. Pour ensuite déclarer une structure il ne faudra pas oublier le mot clé `struct` :

```
struct coord hautgauche, basdroit;
```

En pratique, l'un et l'autre peuvent être utilisés indifféremment. `typedef` donne un code un peu plus concis, mais l'utilisation du mot clé `struct` ne laisse aucun doute sur le type de variable qui a été déclaré.

Résumé

Nous venons d'étudier les structures qui permettent de créer un modèle de données adapté aux besoins de votre programme. Une structure peut contenir tout type de donnée C, y compris des pointeurs, des tableaux ou d'autres structures. Chaque donnée d'une structure, appelée membre, est accessible en associant le nom de la structure et le nom du membre avec l'opérateur (`.`). Les structures sont utilisées individuellement ou en tableaux.

Les unions ne présentent qu'une différence avec les structures : leurs membres sont stockés un par un dans le même emplacement mémoire. On ne peut donc pas en utiliser plusieurs à la fois.

Q & R

Q La définition d'un modèle sans déclaration de structure a-t-elle un sens ?

R Nous avons étudié deux méthodes pour déclarer une structure. La première consiste à définir le modèle de la structure et de le faire suivre d'une occurrence. La seconde consiste à définir le modèle seul, puis à déclarer, plus loin dans le programme, une structure appartenant à ce modèle avec le mot clé struct. C'est une pratique courante en programmation.

Q L'utilisation de typedef est-elle plus fréquente que celle du nom de modèle ?

R Beaucoup de programmeurs utilisent typedef pour alléger leur code source. La plupart des bibliothèques de fonctions utilisent beaucoup typedef pour se différencier. C'est spécialement vrai pour les bibliothèques destinées aux bases de données.

Q Peut-on copier la valeur d'une structure dans une autre avec l'opérateur (=) ?

R Oui et non. Les compilateurs récents permettent d'attribuer les valeurs d'une structure à une autre. Pour les compilateurs plus anciens, il vous faudra attribuer les valeurs membre par membre. La réponse s'applique aussi aux unions.

Q Quelle est la taille d'une union ?

R Un seul emplacement mémoire va recevoir tous les membres de l'union. Sa taille est donc celle du membre le plus encombrant !

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Quelle est la différence entre une structure et un tableau ?
2. Quel est le rôle de l'opérateur (`.`) ?
3. Quel mot clé faut-il utiliser pour créer une structure ?
4. Quelle est la différence entre un nom de modèle de structure et un nom de structure ?
5. Que font les quelques lignes de code suivantes ?

```
struct adresse {
    char nom[31];
    char adr1[31];
    char adr2[31];
    char ville[11];
    char etat[3];
    char zip[11];
} monadresse = { "Bradley Jones",
    "RTSoftware",
    "P.O. Box 1213",
    "Carmel", "IN", "46032-1213"};
```

6. Supposons que vous ayez déclaré un tableau de structures, et que `ptr` soit un pointeur vers le premier élément de ce tableau (c'est-à-dire vers la première structure). Comment faut-il faire pour qu'il pointe sur le second élément ?

Exercices

1. Écrivez la définition d'une structure appelée `time`, contenant trois membres de type `int`.
2. Écrivez le code réalisant les deux tâches suivantes : définition d'une structure `data` contenant un membre de type `int` et deux membres de type `float`, et déclaration d'une structure `info` appartenant au type `data`.
3. Continuez l'exercice 2 en attribuant la valeur 100 au membre de type `int` de la structure `info`.
4. Écrivez la déclaration et l'initialisation d'un pointeur vers `info`.
5. Trouvez deux méthodes, en utilisant le pointeur de l'exercice 4, pour attribuer la valeur 5.5 au premier membre de type `float` de la structure `info`.
6. Écrivez la définition d'une structure appelée `data` qui pourra recevoir une chaîne de 20 caractères.

7. Définissez une structure contenant ces cinq chaînes de caractères : `adresse1`, `adresse2`, `ville`, `etat`, et `zip`. Créez un `typedef RECORD` qui pourra être utilisé pour déclarer des structures appartenant au modèle défini.
8. En utilisant le `typedef` de l'exercice précédent, allouez et initialisez un élément appelé `monadresse`.
9. **CHERCHEZ L'ERREUR :**

```
struct {
    char signe_zodiaque[21];
    int mois;
} signe = "lion", 8;
```

10. **CHERCHEZ L'ERREUR :**

```
/* création d'une union */
union data {
    char un_mot[4];
    long nombre;
}variable_generic = {"WOW", 1000};
```

12

La portée des variables

Dans le Chapitre 5, nous avions abordé le problème de la portée des variables en différenciant celles qui sont définies dans une fonction de celles qui sont définies en dehors de la fonction. Aujourd'hui, vous allez étudier :

- La portée d'une variable et les raisons de son importance
- Les variables externes et les raisons pour lesquelles il vous faudra les éviter
- Les entrées/sorties des variables locales
- La différence entre variables statiques et variables automatiques
- Les variables locales et les blocs
- Ce qui détermine le choix d'une classe de stockage

Définition de la portée

La notion de portée de la variable fait référence aux zones du programme dans lesquelles cette variable est connue, c'est-à-dire aux parties du programme où cette variable est *visible* ou *accessible*. Le terme *variable* utilisé tout au long de ce chapitre englobe tous les types de variables du langage C : variables simples, tableaux, structures, pointeurs, etc. Il représente aussi les constantes symboliques définies avec le mot clé `const`.

Le "temps de vie" de la variable, c'est-à-dire le temps pendant lequel la donnée est conservée en mémoire, dépend aussi de la portée de la variable.

Exemple

Examinons le programme du Listing 12.1. Il définit une variable `x` en ligne 5, utilise `printf()` pour en afficher la valeur à la ligne 11, puis appelle la fonction `print value()` pour afficher de nouveau la valeur de `x`. Vous pouvez remarquer que la fonction `print value` ne reçoit pas `x` en argument, celui-ci est transmis à la fonction `printf()` de la ligne 19.

Listing 12.1 : La variable x est accessible depuis la fonction print_value()

```
 1:  /* Illustration de la portée d'une variable. */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  int x = 999;
 6:
 7:  void print_value(void);
 8:
 9:  int main()
10:  {
11:      printf("%d\n", x);
12:      print_value();
13:
14:      exit(EXIT_SUCCESS);
15:  }
16:
17:  void print_value(void)
18:  {
19:      printf("%d\n", x);
20:  }
```



```
999
999
```

Analyse

La compilation et l'exécution de ce programme ne posent aucun problème. Modifions-le pour que la définition de la variable `x` se retrouve dans la fonction `main()`. Le Listing 12.2 présente le programme modifié.

Listing 12.2 : La variable x n'est pas accessible par la fonction print_value

```
 1:  /* Démonstration de la portée d'une variable. */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  void print_value(void);
 6:
 7:  int main()
 8:  {
 9:      int x = 999;
10:
11:      printf("%d", x);
12:      print_value();
13:
14:      exit(EXIT_SUCCESS);
15:  }
16:
17:  void print_value(void)
18:  {
19:      printf("%d\n", x);
20:  }
```

Analyse

Si vous essayez de compiler ce programme, vous recevrez un message d'erreur similaire à celui-ci :

```
Error  list12_2.c  19 : undefined symbol 'x' in function print_value
```

Le nombre qui suit le nom du fichier est le numéro de la ligne d'où vient l'erreur. La ligne 19 contient l'appel à la fonction `printf()` dans la fonction `print value()`.

Ce message vous indique que, dans la fonction `print value`, la variable `x` n'est pas définie. En d'autres termes, elle n'est pas accessible. Vous pouvez remarquer que l'appel de la fonction `printf()` à la ligne 11 n'a pas généré de message d'erreur, car la variable `x` est visible dans cette partie du programme.

L'unique différence entre le Listing 12.1 et le Listing 12.2 est l'emplacement de la définition de `x`. Cet emplacement détermine sa portée. Dans le Listing 12.1, `x` est une variable *externe* dont la portée s'étend à tout le programme. Les deux fonctions `main()` et `print value()` y ont accès. Dans le Listing 12.2, `x` est une variable *locale* dont la portée

est limitée au seul bloc `main()`. Pour la fonction `print value()`, la variable `x` n'existe pas.

Importance de la notion de portée

L'importance de la notion de portée est liée au principe de la programmation structurée que nous avons étudiée au Chapitre 5. Cette méthode de programmation consiste à diviser le programme en fonctions indépendantes, chacune de ces fonctions réalisant une tâche spécifique. Le mot clé de cette définition est *indépendance*. Pour que cette indépendance soit réelle, la fonction doit posséder ses propres variables, sans possibilité d'interférences avec le reste du programme. La meilleure façon d'obtenir d'une fonction un résultat fiable est d'en isoler les données.

Toutefois, dans certains cas, une isolation complète des données n'est pas souhaitable. Vous apprendrez vite, en tant que programmeur, à jouer sur la portée des variables pour contrôler le niveau "d'isolement" de vos données.

Les variables externes

Une variable *externe* est définie en dehors de toute fonction, y compris de la fonction principale `main()`. La plupart des exemples que nous avons déjà étudiés utilisaient des variables externes définies au début du code source, avant la première exécution de `main()`. Les variables externes sont aussi appelées variables *globales*. Si vous n'initialisez pas une telle variable lors de sa définition, le compilateur le fait de façon implicite avec la valeur 0.

Portée des variables externes

La portée d'une variable externe s'étend au programme tout entier. Elle peut donc être utilisée par la fonction `main()` ou toute autre fonction du programme.

Il existe cependant une restriction. Cette définition n'est vraie que si le code source de votre programme est sauvegardé dans un fichier unique. Vous apprendrez, au Chapitre 21, qu'un programme peut être divisé dans plusieurs fichiers. Il faudra prendre, dans ce cas, des mesures particulières pour les variables externes.

Quand utiliser les variables externes

Les variables externes sont rarement utilisées, car elles vont à l'encontre des principes d'indépendance de la programmation structurée entre les différents blocs du programme. Chaque bloc ou fonction doit contenir le code et les données nécessaires à l'exécution de la tâche qui lui est confiée.

Vous pouvez utiliser une variable externe quand la plupart des fonctions du programme ont besoin d'accéder à une même donnée. Les constantes symboliques définies avec le mot clé `const` sont souvent utilisées de cette façon. Si la donnée ne doit être connue que d'un petit nombre de fonctions, il est préférable de la passer en argument.

Le mot clé *extern*

Quand une fonction utilise une variable externe, il est bon de déclarer cette variable dans la fonction avec le mot clé `extern` :

```
extern type nom;
```

`type` représente le type de la variable dont il précède le nom. Nous pouvons transformer, par exemple, le Listing 12.1 en ajoutant la déclaration de `x` dans les fonctions `main()` et `print value()`. Le programme source obtenu est présenté dans le Listing 12.3.

Listing 12.3 : Déclaration en extern de la variable x dans les fonctions main() et print_value()

```
/* Exemple de déclaration de variables externes. */
#include <stdio.h>
#include <stdlib.h>

int x = 999;

void print_value(void);


int main()
{
    extern int x;

    printf("%d\n", x);
    print_value();

    exit(EXIT_SUCCESS);
}

void print_value(void)
{
    extern int x;
    printf("%d\n", x);
}
```



```
999
999
```

Analyse

Ce programme affiche deux fois la valeur de `x` : d'abord en ligne 13 à partir de la fonction `main()`, puis en ligne 22 dans la fonction `print value()`. La variable `x` est définie ligne 5 avec le type `int` et initialisée avec la valeur 999. Cette variable est déclarée en lignes 11 et 21 avec le type `extern int`. Il faut distinguer la définition de la variable, qui réserve un emplacement mémoire pour cette variable, et la déclaration `extern`. Cette dernière signifie : "cette fonction utilise la variable externe de tel type, et de tel nom, qui est définie dans une autre partie du programme". Dans notre exemple, la déclaration de la ligne 21 n'est pas indispensable. Toutefois, si la fonction `print value` s'était trouvée dans un bloc de code différent de celui de la déclaration globale de la ligne 5, cette déclaration `extern` aurait été obligatoire.

Les variables locales

Une *variable locale* est une variable définie dans une fonction, et sa portée se limite à la fonction dans laquelle elle est définie. Nous avons étudié ces variables dans le Chapitre 5. Contrairement aux variables globales, celles-ci ne sont pas initialisées par le compilateur lorsque vous omettez de le faire. Une variable locale qui n'a pas été initialisée contient une valeur indéterminée.

La variable `x` du Listing 12.2 est une variable locale pour la fonction `main()`.



À faire

Utiliser des variables locales pour les compteurs de boucle.

Utiliser des variables locales pour isoler les valeurs qu'elles contiennent du reste du programme.

À ne pas faire

Utiliser des variables externes si elles ne sont pas utilisées par la majorité des fonctions du programme.

Variables statiques et automatiques

Par défaut, les variables locales sont automatiques. Cela signifie qu'elles sont créées et détruites avec l'appel et la fin de la fonction. En d'autres termes, la valeur de cette variable n'est pas conservée entre deux appels de la fonction dans laquelle elle est définie.

Si la dernière valeur de la variable locale doit être connue à l'appel de la fonction, la variable doit être définie avec le mot clé `static`. Une fonction d'impression, par exemple, doit connaître le nombre de lignes déjà envoyées vers l'imprimante pour effectuer correctement le changement de page. Voici un exemple de définition d'une variable statique :

```
void fonc1(int x)
{
    static int a;
.../*Instructions */
}
```

Le Listing 12.4 illustre la différence entre variables locales statiques et variables locales automatiques.

Listing 12.4 : Différence entre variables statiques et variables automatiques

```
/* Exemple de variables locales statiques et automatiques. */
#include <stdio.h>
#include <stdlib.h>

void func1(void);

int main()
{
    int count;

    for (count = 0; count < 20; count++)
    {
        printf("Iteration n° %d: ", count);
        func1();
   }

   exit(EXIT_SUCCESS);
}

void func1(void)
{
   static int x = 0;
   int y = 0;

   printf("x = %d, y = %d\n", x++, y++);
}
```



```
Iteration n° 0 : x = 0, y = 0
Iteration n° 1 : x = 1, y = 0
Iteration n° 2 : x = 2, y = 0
Iteration n° 3 : x = 3, y = 0
Iteration n° 4 : x = 4, y = 0
Iteration n° 5 : x = 5, y = 0
```

```
Iteration n° 6 : x = 6, y = 0
Iteration n° 7 : x = 7, y = 0
Iteration n° 8 : x = 8, y = 0
Iteration n° 9 : x = 9, y = 0
Iteration n° 10 : x = 10, y = 0
Iteration n° 11 : x = 11, y = 0
Iteration n° 12 : x = 12, y = 0
Iteration n° 13 : x = 13, y = 0
Iteration n° 14 : x = 14, y = 0
Iteration n° 15 : x = 15, y = 0
Iteration n° 16 : x = 16, y = 0
Iteration n° 17 : x = 17, y = 0
Iteration n° 18 : x = 18, y = 0
Iteration n° 19 : x = 19, y = 0
```

Analyse

Ce programme contient la fonction `func1()` qui définit et initialise une variable de chaque type (lignes 20 à 26). À chaque appel de la fonction, les variables sont affichées et incrémentées (ligne 25). La fonction principale `main()` (lignes 7 à 18) contient une boucle `for (lignes 11 à 15)`, qui envoie un message à l'écran (ligne 13) et appelle la fonction `func1()` (ligne 14). Cette boucle s'exécute 20 fois.

Le résultat de l'exécution du programme vous montre que la valeur de la variable statique `x` augmente à chaque exécution de la boucle, car la valeur est conservée entre deux appels. La variable automatique `y`, au contraire, est initialisée à 0 à chaque appel de la fonction.

Ce programme illustre aussi la différence de traitement des deux initialisations d'une variable (lignes 22 et 23). La variable statique n'est initialisée qu'au premier appel de la fonction. Quand la fonction est de nouveau appelée, le programme se souvient que cette variable a déjà été initialisée ; il ne va donc pas recommencer l'opération. La variable gardera ainsi sa valeur antérieure. La variable automatique, au contraire, est initialisée à chaque appel de la fonction.

Automatique étant le type par défaut pour une variable locale, il n'est pas nécessaire de l'indiquer dans la définition. Vous pouvez tout de même inclure le mot clé `auto` de cette façon :

```
void func1(int y)
{
    auto int count;
/* instructions ... */
}
```

La portée des paramètres d'une fonction

Les variables de la liste des paramètres d'une fonction ont une portée locale. Étudions l'exemple suivant :

```
void func1(int x)
{
    int y;
/* instructions ... */
}
```

`x` et `y` sont des variables locales pour la fonction `func1()`. La valeur initiale de `x` dépend de la valeur transmise par le programme appelant. Quand la fonction a utilisé cette valeur, `x` peut être traitée comme n'importe quelle variable locale.

Les variables paramètres étant toujours initialisées avec la valeur passée en argument par le programme appelant, les termes statique ou automatique n'ont pas de sens en ce qui les concerne.

Les variables statiques externes

On peut donner le type statique à une variable externe en ajoutant le mot clé `static` dans sa définition :

```
static float taux;
int main()
{
/* instructions ... */
}
```

La différence entre une variable externe et une variable externe statique concerne la portée de ces variables. Une variable externe statique est visible par toutes les fonctions du fichier dans lequel elle se trouve. Une variable externe simple est visible par toutes les fonctions du fichier et peut être utilisée par des fonctions dans d'autres fichiers.

La répartition d'un code source dans des fichiers distincts est traitée dans le Chapitre 21.

Variables de type register

Le mot clé `register` est utilisé pour demander au compilateur de stocker une variable locale automatique dans un registre plutôt que dans un emplacement de la mémoire standard.

Le processeur central (CPU) de votre ordinateur contient quelques emplacements mémoire appelés *registres*. Ces registres sont utilisés pour des opérations comme l'addition ou la division. La CPU prend les données en mémoire, les copie dans les registres,

réalise l'opération demandée, puis les replace en mémoire. Si une variable est enregistrée directement dans un registre, les opérations qui la concernent seront effectuées plus rapidement. Exemple :

```
void func1(void)
{
    register int x;
/* instructions ... */
}
```

Le mot clé `register` effectue une demande au compilateur, et non un ordre. En fonction des besoins du programme, le registre pourra ne pas être disponible pour la variable. Dans ce cas, le compilateur traitera la variable comme une variable automatique ordinaire. Le type de stockage `register` doit être choisi pour les variables souvent utilisées par la fonction, pour un compteur de boucle par exemple.

Le mot clé `register` ne s'applique qu'aux variables numériques simples. On ne peut pas l'utiliser avec les tableaux ou les structures. De même, il ne peut être utilisé avec les classes de stockage externe ou statique, et vous ne pouvez pas définir un pointeur vers une variable de type `register`.

Enfin, et c'est peut-être le point le plus important, le mot clé `register` ne devrait plus être utilisé pour les variables de programmes destinés à être utilisés sur les machines puissantes d'aujourd'hui. En effet, les processeurs sont devenus complexes et il vaut mieux faire confiance aux compilateurs qui savent mieux optimiser que la plupart des programmeurs.



À faire

Initialiser les variables locales pour être sûr de la valeur qu'elles contiennent.

Initialiser les variables globales, même si elles le sont par défaut. En prenant l'habitude de toujours initialiser vos variables, vous éviterez des erreurs.

Transmettre les données en tant que paramètres d'une fonction plutôt que les déclarer comme variables globales si elles ne sont utilisées que par quelques fonctions.

À ne pas faire

Utiliser le type de variable `register` pour des valeurs non numériques, des structures ou des tableaux.

Les variables locales et la fonction *main()*

La fonction `main()` est appelée quand le programme commence son exécution, et rend le contrôle au système d'exploitation quand le programme est terminé.

Cela signifie qu'une variable locale définie dans `main()` est créée quand le programme commence, et qu'elle cesse d'exister quand le programme se termine. Pour cette fonction, la notion de variable locale statique n'a donc aucun sens. Une variable ne peut subsister entre deux exécutions du programme. Il n'y a donc aucune différence pour la fonction `main()` entre une variable locale statique et une variable locale automatique.

Choix de la classe de stockage

Le Tableau 12.1 présente les cinq classes de stockage disponibles en C. Il vous aidera à faire votre choix.

Tableau 12.1 : Les cinq classes de stockage pour les variables du langage C

Classe de stockage	*Mot clé*	*Durée de vie*	*Définition*	*Portée*
Automatique	Aucun [1]	Temporaire	Dans la fonction	Locale
Statique	static	Temporaire	Dans la fonction	Locale
Registre	register	Temporaire	Dans la fonction	Locale
Externe	Aucun [2]	Permanent	Hors de la fonction	Globale (tous les fichiers)
Externe	static	Permanent	Hors de la fonction	Globale (un fichier)

1. Le mot clé `auto` est en option
2. Le mot clé `extern` est utilisé dans les fonctions pour déclarer une variable externe statique qui est définie ailleurs dans le programme.

Quand vous choisissez votre classe de stockage, préférez une classe automatique chaque fois que c'est possible, et utilisez les autres seulement quand c'est nécessaire. Voici quelques règles pour vous guider :

- Commencez en donnant une classe de stockage locale automatique à chaque variable.
- Pour toutes les fonctions, sauf `main()`, choisissez la classe statique quand la valeur de la variable doit être conservée entre deux appels.
- Si la variable est utilisée par toutes ou presque toutes les fonctions, définissez-la avec la classe externe.

Variables locales et blocs

Nous n'avons parlé que de variables locales pour une fonction, mais vous pouvez définir des variables locales pour un bloc du programme (portion de code entouré par des accolades). Le Listing 12.5 vous en présente un exemple.

Listing 12.5 : Définition de variables locales dans un bloc du programme

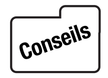
```
 1:  /* Exemple de variable locale dans un bloc. */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  int main()
 6:  {
 7:       /* Définition d'une variable locale pour main(). */
 8:
 9:          int count = 0;
10:
11:         printf("Hors du bloc, count = %d\n", count);
12:
13:         /* Début d'un bloc. */
14:      {
15:         /* Définition d'une variable locale pour le bloc. */
16:
17:         int count = 999;
18:         printf("Dans le bloc, count = %d\n", count);
19:      }
20:
21:      printf("De nouveau hors du bloc, count = %d\n", count);
22:      exit(EXIT_SUCCESS);
23:  }
```

```
Hors du bloc, count = 0
Dans le bloc, count = 999
De nouveau hors du bloc, count = 0
```

Analyse

Ce programme vous démontre que la variable `count` dans le bloc est indépendante de la variable `count` définie en dehors du bloc. La ligne 9 définit la variable `count` de type `int` et l'initialise à 0. La portée de cette variable est la fonction `main()`. La ligne 11 affiche la valeur d'initialisation de la variable (`0`). Les lignes 14 à 19 représentent un bloc dans lequel une autre variable `count` est définie avec le type `int`. La ligne 18 affiche la valeur d'initialisation de cette variable : 999. L'instruction `printf()` de la ligne 21 se trouvant après la fin du bloc (ligne 19) ; elle utilise la variable `count` initiale (déclarée en ligne 9) de la fonction `main()`.

L'emploi de ce type de variable locale n'est pas courant en programmation C. Son utilité est de permettre à un programmeur d'isoler un problème dans un programme. Il suffit de découper le code en blocs avec des accolades, et d'introduire des variables locales pour trouver l'erreur.



À faire

Positionner des variables en début de bloc (temporairement) pour identifier un problème.

À ne pas faire

Placer une définition de variable dans un programme ailleurs qu'en début de fonction ou de bloc (contrairement au C++).

Résumé

L'étude de ce chapitre vous a fait découvrir les classes de stockage des variables du langage C. Toute variable C, de la variable simple aux structures en passant par les tableaux, a une classe de stockage spécifique. Cette classe détermine deux caractéristiques : la portée, ou de quelle partie du programme la variable est visible, et la durée de vie de la variable en mémoire.

Le choix de la classe de stockage est un aspect important de la programmation structurée. En utilisant dans les fonctions un maximum de variables locales, vous rendez ces fonctions indépendantes les unes des autres. Une variable doit appartenir à une classe de stockage automatique, sauf s'il existe une raison particulière de la rendre externe ou statique.

Q & R

Q Pourquoi ne pas toujours utiliser des variables globales puisqu'elles peuvent être utilisées n'importe où dans le programme ?

R Quand vos programmes commenceront à atteindre une taille respectable, ils contiendront de plus en plus de déclarations de variables. La mémoire disponible sur votre machine n'est pas illimitée. Les variables globales occupent une place en mémoire pendant toute la durée d'exécution du programme, alors que les variables locales n'occupent un emplacement mémoire que pendant la durée d'exécution de leur fonction. (Une variable statique conserve son emplacement mémoire depuis sa première

utilisation jusqu'à la fin de l'exécution du programme.) De plus, la valeur d'une variable globale peut être altérée accidentellement par une des fonctions. Elle ne contiendra donc plus la bonne valeur quand vous en aurez besoin.

Q Nous avons vu, au Chapitre 11, que la portée influence une structure, mais pas son modèle. Pourquoi ?

R Quand vous déclarez une structure sans occurrences, vous créez un modèle ; il n'y a pas de déclaration de variable. C'est la raison pour laquelle le modèle peut se situer en dehors de toute fonction sans effet réel sur la mémoire. Beaucoup de programmeurs sauvegardent leurs modèles de structures dans les fichiers en-tête. Il ne leur reste plus qu'à inclure ce fichier quand il leur faut créer une structure appartenant à un des modèles. (Les fichiers en-tête sont traités dans le Chapitre 21.)

Q Comment l'ordinateur fait-il la différence entre une variable globale et une variable locale qui auraient le même nom ?

R Quand une variable locale est déclarée avec le même nom qu'une variable globale, cette dernière est ignorée temporairement par le programme (jusqu'à ce que la variable locale ne soit plus visible).

Atelier

Cet atelier comporte un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Qu'est-ce que la portée ?
2. Quelle est la principale différence entre une classe de stockage locale et une classe de stockage externe ?
3. En quoi l'emplacement de la définition d'une variable affecte-t-il la classe de stockage ?
4. Lorsqu'on définit une variable locale, quelles sont les deux options qui concernent la durée de vie de cette variable ?
5. Quand elles ont été définies, votre programme peut initialiser des variables locales statiques et automatiques. Quand doit-on faire ces initialisations ?
6. Vrai ou faux : une variable de type `register` sera toujours stockée dans un registre.

7. Quelle est la valeur contenue dans une variable globale qui n’a pas été initialisée ?
8. Quelle est la valeur contenue dans une variable locale qui n’a pas été initialisée ?
9. Quel sera le message affiché par la ligne 21 du Listing 12.5 si les lignes 9 et 11 sont supprimées ?
10. Comment doit-on déclarer une variable locale de type `int` pour que sa valeur soit conservée entre deux appels de la fonction qui l’utilise ?
11. À quoi sert le mot clé `extern` ?
12. À quoi sert le mot clé `static` ?

Exercices

1. Corrigez l’erreur du Listing 12.2 sans utiliser de variable externe.
2. Écrivez un programme qui déclare une variable globale `var` de type `int`. Initialisez cette valeur puis affichez son contenu en utilisant une fonction autre que `main()`. Est-il nécessaire de transmettre `var` dans la liste des paramètres de la fonction ?
3. Transformez le programme de l’exercice 3 pour que la variable `var` soit une variable locale de la fonction `main()`. Est-il maintenant nécessaire de transmettre `var` dans la liste des paramètres de la fonction ?
4. Un programme peut-il avoir une variable locale et une variable globale du même nom ? Écrivez un programme pour justifier votre réponse.
5. **CHERCHEZ L’ERREUR :** Pourrez-vous trouver le problème de ce code ? (Conseil : l’erreur vient de l’endroit où une variable est déclarée.)

```
void exemple_de_fonction(void)
{
    int ctr1;
    for (ctr1 = 0; ctr1 < 25; ctr1++)
      printf("*");
    puts ("Cela est un exemple de fonction\n");
    {
      char star = '*';
      puts( "il y a un problème\n" );
      for (int ctr2 = 0; ctr2 < 25; ctr2++)
      {
        printf("%c", star);
      }
    }
}
```

6. **CHERCHEZ L'ERREUR :**

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x = 1;
    static int tally = 0;
    for (x = 0; x < 101; x++)
    {
      if (x % 2 == 0)    /*si x est pair… */
         tally++;..    /* on ajoute 1 à tally. */
    }
    printf("Il y a %d nombres pairs.\n", tally);
   exit(EXIT_SUCCESS);
}
```

7. **CHERCHEZ L'ERREUR :**

```
#include <stdio.h>
#include <stdlib.h>
void print_fonction(char star);
int ctr;
int main()
{
    char star;
    print_fonction(star);
    exit(EXIT_SUCCESS);
}
void print_fonction (char star)
{
    char dash;
    for (ctr = 0; ctr < 25; ctr++)
    {
      printf("%c%c", star, dash);
    }
}
```

8. Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
void print_letter2(void);        /* Déclaration de la fonction */
int ctr;
char letter1 = 'X';
char letter2 = '=';
int main()
{
    for(ctr = 0; ctr < 10; ctr++)
    {
```

```
      printf("%c", letter1);
      print_letter2();
    }
   exit(EXIT_SUCCESS);
}
void print_letter2(void)
{
    for(ctr = 0; ctr < 2; ctr++)
      printf("%c", letter2);
}
```

9. **CHERCHEZ L'ERREUR :** Le programme précédent peut-il être exécuté ? Si la réponse est non, quel est le problème ? Corrigez-le.

Exemple pratique 4

Les messages secrets

Voici la quatrième section de ce type. Le programme qu'elle présente comprend de nombreux éléments étudiés précédemment, et en particulier les fichiers disques traités au Chapitre 16.

Le programme qui suit permet de coder ou de décoder des messages. Pour l'exécuter, vous devrez fournir deux paramètres :

```
coder nomfichier action
```

nomfichier représente soit le nom du fichier à créer pour enregistrer le message secret, soit le nom du fichier qui contient le message à décoder. L'*action* sera `D` pour décoder ou C pour coder un message. Si vous lancez le programme sans lui transmettre de paramètre, il affichera les instructions pour taper correctement la commande.

En transmettant ce programme à des amis ou connaissances, vous pourrez leur envoyer des messages codés. Ceux-ci ne pourront être lus que par l'intermédiaire du programme !

Listing Exemple pratique 4 : coder.c

```
/* Programme : coder.c
 * Syntaxe   : coder [nomfichier] [action]
 *         nomfichier est le nom du fichier pour les données codées
 *         action est égal à D pour décoder, ou n'importe quel
 *         autre caractère pour coder
 *---------------------------------------------------------------*/

 #include <stdio.h>
 #include <stdlib.h>
```

```
#include <string.h>

int encode_character( int ch, int val );
int decode_character( int ch, int val );

int main( int argc, char *argv[])
{
    FILE *fh;             /* Descripteur du fichier  */
    int rv = 1;           /* valeur renvoyée */
    int ch = 0;           /* variable pour stocker un caractère */
    unsigned int ctr = 0; /* compteur */
    int val = 5;          /* valeur pour coder avec */
    char buffer[257];     /* le buffer */

    if( argc != 3 )
    {
        printf("\nErreur:  nombre de paramètres incorrect..." );
        printf("\n\nSyntaxe:\n   %s nomfichier action", argv[0]);
        printf("\n\n   Où:");
        printf("\n        nomfichier = nom du fichier à coder ");
        printf("ou à décoder");
        printf("\n        action   = D pour décoder ou C pour coder\n\n");
        rv = -1;       /* valeur de l'erreur renvoyée */
    }
    else
    if(( argv[2][0] == 'D') || (argv[2][0] == 'd' )) /* décodage */
    {   fh = fopen(argv[1], "r");   /* ouverture du fichier   */
        if( fh <= 0 )               /* contrôle  */
        {
            printf( "\n\nErreur d'ouverture du fichier..." );
            rv = -2;             /* valeur de l'erreur renvoyée */
        }
        else
        {
            ch = getc( fh );     /* lecture d'un caractère */
            while( !feof( fh ) )  /* Fin du fichier? */
            {
               ch = decode_character( ch, val );
               putchar(ch);  /* Affichage du caractère */
               ch = getc( fh);
            }

            fclose(fh);
            printf( "\n\nFichier décodé et affiché.\n" );
        }
    }
    else  /* Codage dans un fichier. */
    {

        fh = fopen(argv[1], "w");
        if( fh <= 0 )
        {
            printf( "\n\nErreur pendant la création du fichier..." );
            rv = -3;  /* Valeur renvoyée */
        }
```

```
65:         else
66:         {
67:             printf("\n\nEntrez le texte à coder. ");
68:             printf("Entrez une ligne vide pour terminer.\n\n");
69:
70:             while( fgets(buffer, sizeof(buffer), stdin))
71:             {
72:                 if((buffer[0] == 0) || (buffer[0] == '\n'))
73:                      break;
74:
75:                 for( ctr = 0; ctr < strlen(buffer); ctr++ )
76:                 {
77:                     ch = encode_character( buffer[ctr], val );
78:                     ch = fputc(ch, fh);  /* Ecriture du fichier */
79:                 }
80:             }
81:             printf( "\n\nMessage codé et enregistré.\n" );
82:             fclose(fh);
83:         }
84:
85:     }
86:     exit((rv==1)?EXIT_SUCCESS:EXIT_FAILURE);
87: }
88:
89: int encode_character( int ch, int val )
90: {
91:    ch = ch + val;
92:    return (ch);
93: }
94:
95: int decode_character( int ch, int val )
96: {
97:     ch = ch - val;
98:     return (ch);
99: }
```

Voici un exemple de message secret :

```
hjhn%jxy%zs%rjxxflj%hti‡
```

Il signifie :

```
Ceci est un message codé
```

Analyse

Le programme code et décode simplement en ajoutant ou en soustrayant une valeur aux caractères. Le code obtenu sera bien sûr très facile à déchiffrer. Vous pouvez compliquer un peu ce chiffrage en remplaçant les lignes 91 et 97 par la ligne suivante :

```
ch = ch ^ val;
```

L'opérateur binaire mathématique ^ modifie les bits du caractère. Le codage obtenu sera plus difficilement déchiffré.

Si vous envisagez de distribuer ce programme à plusieurs personnes différentes, vous pourriez ajouter un troisième paramètre pour définir `val`. Cette variable contient la valeur utilisée pour le codage ou le décodage.



Ce programme est loin d'être blindé (par exemple `val` *doit être inférieur à 13, valeur du code ASCII du retour à la ligne). L'algorithme utilisé est de plus extrêmement simple et ne résistera pas longtemps à une personne malveillante ou trop curieuse. Si vous avez un réel besoin de chiffrer vos données, orientez-vous plutôt vers un logiciel du marché tel que le logiciel libre gpg. Sachez également que le chiffrage de données est soumis à la législation en vigueur.*

13

Les instructions de contrôle (suite)

Le Chapitre 6 a introduit quelques-unes des instructions de contrôle du langage C. Ces instructions permettent de contrôler l'exécution des autres instructions du programme. Ce chapitre couvre d'autres aspects du contrôle des programmes, comme l'instruction `goto`, et quelques exemples intéressant de ce que l'on peut faire avec une boucle. Aujourd'hui, vous allez étudier :

- Les instructions `break` et `continue`
- La définition et l'intérêt des boucles continues
- Le fonctionnement de `goto`
- L'instruction `switch`
- Les différentes manières de sortir du programme
- L'exécution automatique de fonctions une fois le programme terminé
- L'introduction de commandes système dans votre programme

Fin de boucle prématurée

Nous avons étudié, au Chapitre 6, les trois instructions de boucle `for`, `while` et `do while`. Ces boucles permettent d'exécuter un bloc d'instructions de zéro à *n* fois, en fonction de conditions établies dans le programme. Dans tous les cas, la sortie de la boucle n'intervient que lorsqu'une certaine condition est remplie.

Les deux instructions que nous allons maintenant étudier, `break` et `continue`, permettront d'exercer un contrôle supplémentaire sur l'exécution de ces trois boucles.

L'instruction *break*

Cette instruction se place exclusivement dans le corps d'une boucle `for`, `while` ou `do while` (ou avec l'instruction `switch` étudiée en fin de chapitre). Quand une instruction `break` est rencontrée en cours d'exécution du programme, la sortie de la boucle est immédiate. Exemple :

```
for (count = 0; count < 10; count++)
{
    if (count == 5)
       break;
}
```

La première instruction est celle d'une boucle qui doit s'exécuter 10 fois. Pourtant, à la sixième itération, la variable `count` est égale à 5 et l'instruction `break` s'exécute provoquant la fin de la boucle `for`. L'exécution se poursuit avec la première instruction qui suit l'accolade de fin de la boucle. Quand cette instruction `break` se trouve dans une boucle imbriquée, l'exécution se poursuit avec la suite de la première boucle.

Listing 13.1 : Utilisation de l'instruction break

```
/* Exemple d'utilisation de l'instruction break. */
#include <stdio.h>
#include <stdlib.h>

char s[] = "Cela est une chaîne de test. Elle contient deux \
            phrases.";
int main()
{
    int count;

    printf("Chaîne initiale :  %s\n", s);

    for (count = 0; s[count]!='\0'; count++)
    {
        if (s[count] == '.')
        {
```

```
17:             s[count+1] = '\0';
18:             break;
19:         }
20:     }
21:     printf("Chaîne modifiée :  %s\n", s);
22:
23:     exit(EXIT_SUCCESS);
24: }
```



```
Chaîne initiale : Cela est une chaîne de test. Elle contient deux phrases.
Chaîne modifiée : Cela est une chaîne de test.
```

Analyse

Ce programme extrait la première phrase d'une chaîne de caractères. La boucle `for` (lignes 13 à 20) analyse la chaîne, caractère par caractère, pour trouver le premier point. La variable `count` est initialisée et incrémentée à chaque exécution de la boucle pour se positionner sur le caractère suivant de la chaîne `s`. La ligne 15 vérifie si le caractère pointé est un point. Si c'est le cas, la ligne 17 ajoute le caractère de fin de chaîne immédiatement après le point. La chaîne initiale étant ainsi tronquée, il n'est plus nécessaire de poursuivre l'exécution la boucle, qui est alors interrompue par l'instruction `break`. L'exécution du programme se poursuit en ligne 21. Si la boucle ne trouve pas de point, la chaîne reste inchangée.

Une boucle peut contenir plusieurs instructions `break`, mais seule la première rencontrée sera exécutée. Si aucune instruction `break` n'est exécutée, la boucle se termine normalement. La Figure 13.1 vous montre le mécanisme de cette instruction.

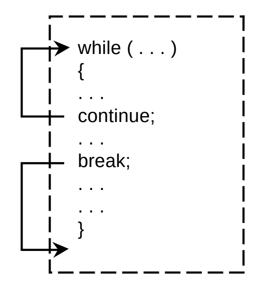


Figure 13.1
Les instructions break et continue.

Syntaxe de l'instruction *break*

```
break;
```

Elle est utilisée à l'intérieur d'une boucle ou d'une instruction `switch`. Elle provoque l'arrêt de la boucle ou de l'instruction `switch` en cours, et donne le contrôle à l'instruction située après la fin de cette boucle ou de cette instruction `switch`.

Exemple

```
int x;
printf("comptons de 1 à 10\n");
/* La boucle ne contient pas de condition ! */
for (x = 1; ; x++)
{
    if (x == 10)    /* recherche de la valeur 10 */
    break;    /* fin de la boucle */
    printf("\n%d", x);
}
```

L'instruction continue

Comme l'instruction `break`, `continue` ne peut être placée que dans le corps d'une boucle `for`, `while`, ou `do while`. L'exécution d'une instruction `continue` arrête le processus de bouclage, et l'exécution recommence en début de boucle. La Figure 13.1 donne le schéma de fonctionnement de cette instruction.

Le Listing 13.2 utilise l'instruction `continue`. Il lit une ligne de texte entrée au clavier pour l'afficher ensuite sans ses voyelles.

Listing 13.2 : Utilisation de l'instruction continue

```
/* Exemple d'utilisation de l'instruction continue. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    /* Déclaration d'une mémoire tampon pour les données entrées, */
    /* et d'un compteur. */
    char buffer[81];
    int ctr;

    /* Lecture d'une ligne de texte. */

    puts("Entrez une ligne de texte : ");
    lire_clavier(buffer, sizeof(buffer));

    /* On se déplace dans la chaîne en affichant tous les */
    /* caractères qui ne sont pas des voyelles minuscules. */

    for (ctr = 0; buffer[ctr] !='\0'; ctr++)
    {

        /* Si le caractère est une voyelle minuscule, */
        /* il n'est pas affiché. */
        if (buffer[ctr] == 'a' || buffer[ctr] == 'e'
```

```
26:          || buffer[ctr] == 'i' || buffer[ctr] == 'o'
27:          || buffer[ctr] == 'u')
28:              continue;
29:          /* Si ce n'est pas une voyelle, on l'affiche. */
30:
31:          putchar(buffer[ctr]);
32:      }
33:
34:      exit(EXIT_SUCCESS);
35:  }
```



```
Entrez une ligne de texte :
Cela est une ligne texte
Cl st n lgn d txt
```

Analyse

Ce programme n’a pas beaucoup d’intérêt pratique, mais il utilise une instruction `continue`. Les variables sont déclarées en lignes 9 et 10. Le tableau `buffer[]` stocke la ligne de texte lue en ligne 15. La variable `ctr` permet de se déplacer d’un élément à l’autre de `buffer[]`, pendant que la boucle `for` (lignes 20 à 32) cherche les voyelles. Pour cela, l’instruction `if` (lignes 25 à 27) compare chaque lettre de la chaîne avec les cinq voyelles minuscules. Si la lettre correspond à une voyelle, l’instruction `continue` est exécutée et le contrôle est donné à la ligne 20 du programme. Si la lettre n’est pas une voyelle, l’exécution se poursuit avec l’instruction `if` de la ligne 31. `putchar()` est une fonction de la bibliothèque qui permet d’afficher un caractère à l’écran.

Syntaxe de l’instruction *continue*

```
continue;
```

L’instruction `continue` doit être utilisée dans une boucle. Elle provoque l’exécution immédiate de la prochaine itération de la boucle. Les instructions situées entre `continue` et la fin de la boucle sont ignorées.

Exemple

```
int x;
printf("On n'affiche que les nombres pairs entre 1 et 10\n");
for(x = 1; x<= 10; x++)
{
    if(x % 2 != 0)    /* contrôle de la parité */
         continue;    /* on récupère la prochaine occurrence de x */
    printf("\n%d", x);
```

L’instruction goto

Cette instruction fait partie des instructions du langage C qui exécutent un *saut inconditionnel* ou un *branchement*. Rencontrée au cours de l’exécution d’un programme, une instruction `goto` provoque le transfert ou le branchement immédiat vers l’emplacement désigné. Ce branchement est dit inconditionnel, car il ne dépend d’aucune condition.

L’instruction `goto` et son étiquette peuvent se trouver dans des blocs de code différents, mais doivent toujours faire partie de la même fonction. Le Listing 13.3 présente un programme simple utilisant `goto`.

Listing 13.3 : Utilisation de l’instruction goto

```
/* Illustration de l'instruction goto */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n;

start: ;

    puts("Entrez un nombre entre 0 et 10: ");
    scanf("%d", &n);

    if (n < 0 || n > 10)
      goto start;
    else if (n == 0)
      goto localisation0;
    else if (n == 1)
      goto localisation1;
    else
      goto localisation2;

localisation0: ;
    puts("Vous avez tapé 0.\n");
      goto end;

localisation1: ;
    puts("Vous avez tapé 1.\n");
      goto end;

localisation2: ;
    puts("Vous avez tapé quelque chose entre 2 et 10.\n");

end: ;
    return 0;

```

```
37:  }
Entrez un nombre entre 0 et 10 :
9
Vous avez tapé quelque chose entre 2 et 10.
```



À faire

Utiliser `goto` à bon escient, en particulier pour se brancher sur des instructions de traitement d'erreur lorsqu'une erreur est survenue.

Utiliser des boucles (`while`, `for`...), break ou continue lorsqu'on peut éviter d'utiliser `goto`.

À ne pas faire

Confondre `break` et `continue` : la première termine une boucle prématurément, alors que la seconde relance la boucle pour l'itération suivante.

Utiliser `goto` à tort et à travers.

Analyse

Ce programme simple lit un nombre compris entre 0 et 10. Si l'utilisateur entre un nombre n'appartenant pas à cet intervalle de valeurs, l'instruction `goto` de la ligne 15 donne le contrôle du programme à la ligne 9. Cette ligne est celle de l'étiquette `start`. Si le nombre lu est bien compris entre 0 et 10, la ligne 16 compare sa valeur avec 0. Si sa valeur est égale à 0, l'instruction `goto` de la ligne 17 donne le contrôle à la ligne 23 (`localisation0`). Un message est alors envoyé à l'utilisateur (ligne 24) et une autre instruction `goto` est exécutée. Cette dernière instruction se branche sur l'étiquette `end` qui est la fin du programme. Le programme suit la même logique pour les autres chiffres.

L'étiquette associée à l'instruction `goto` peut se situer avant ou après cette instruction dans le code, l'essentiel étant qu'elles soient toutes les deux dans la même fonction. Elles peuvent se trouver dans des blocs différents : vous pouvez coder, par exemple, un transfert de l'intérieur d'une boucle vers l'extérieur au moyen d'une instruction `for`. Toutefois, nous vous recommandons fortement de ne jamais utiliser `goto` dans vos programmes. Voici pourquoi :

- Vous n'avez pas besoin de `goto`. Toutes les tâches que vous aurez à programmer peuvent être réalisées avec les autres instructions de branchement du langage C.
- C'est dangereux. Quand un programme exécute un branchement après une instruction `goto`, il ne garde aucune trace de l'emplacement d'où il vient et l'ordre d'exécution va vite se compliquer. C'est ce que l'on appelle le *code spaghetti*.

Syntaxe de l'instruction *goto*

```
goto identifiant;
```

L'identifiant est une instruction label qui identifie un emplacement du programme pour la suite de l'exécution. Une instruction label est constituée d'un identifiant suivi de deux points, puis d'une instruction C :

```
identifiant: instruction C ;
```

Si vous ne voulez indiquer que le label, vous pouvez le faire suivre de l'instruction nulle :

```
localisation1: ;
```

Les boucles infinies

Une boucle infinie est une boucle `for`, `while` ou `do while`, qui bouclerait toujours si on ne lui ajoutait pas des instructions. En voici un exemple :

```
while (1)
{
    /* instructions... */
}
```

La condition que teste `while` est la constante 1, qui sera toujours vraie et ne pourra pas être changée par le programme. Ainsi, la boucle `while` ne s'arrêtera jamais d'elle-même.

Le contrôle de cette boucle s'obtient avec l'instruction `break` sans laquelle ce type de boucle serait inutile.

Vous pouvez aussi créer des boucles infinies `for` ou `do while` :

```
for (;;)
{
    /* instructions...*/
}
do
{
    /* instructions...*/
} while (1);
```

Ces trois boucles suivent le même principe. Nous avons choisi, pour nos exemples, d'utiliser une boucle `while`.

L'intérêt d'une boucle infinie est que l'on peut faire de nombreux tests de conditions avant de décider de l'arrêt de cette boucle. En effet, il aurait été difficile d'inclure tous ces tests entre

les parenthèses de l'instruction `while`. Il est plus facile de les effectuer séparément dans le corps de la boucle, puis de sortir avec une instruction `break`.

Une boucle infinie peut permettre de créer un menu système pour orienter les opérations réalisées par votre programme. Le Listing 13.4 vous en présente un exemple.

Listing 13.4 : Réalisation d'un menu système avec une boucle infinie

```
/* Réalisation d'un menu système avec une boucle infinie. */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int menu(void);

int main()
{
    int choix;

    while (1)
    {

/* Lecture du choix de l'utilisateur. */
        choix = menu();

/* Le branchement est réalisé en fonction du choix. */

        if (choix == 1)
        {
            puts("\nExécution de la tâche correspondant au choix 1.");
            sleep(5);
        }
        else if (choix == 2)
        {
            puts("\nExécution de la tâche correspondant au choix 2.");
            sleep(5);
        }
        else if (choix == 3)
        {
            puts("\nExécution de la tâche correspondant au choix 3.");
            sleep(5);
        }
        else if (choix == 4)
        {
            puts("\nExécution de la tâche correspondant au choix 4.");
            sleep(5);
        }
        else if (choix == 5)   /* Sortie du programme. */
        {
            puts("\nSortie du programme...\n");
            sleep(5);
            break;
```

Listing 13.4 : Réalisation d'un menu système avec une boucle infinie *(suite)*

```
        }
        else
        {
            puts("\nChoix incorrect, essayez de nouveau.");
            sleep(5);
        }
    }
    exit(EXIT_FAILURE);

}

/* Affichage du menu et lecture du choix de l'utilisateur. */
int menu(void)
{
    int reponse;

    puts("\nEntrez 1 pour la tâche A.");
    puts("Entrez 2 pour la tâche B.");
    puts("Entrez 3 pour la tâche C.");
    puts("Entrez 4 pour la tâche D.");
    puts("Entrez 5 pour sortir du programme.");

    scanf("%d", &reponse);

    return reponse;
}
```

```
Entrez 1 pour la tâche A.
Entrez 2 pour la tâche B.
Entrez 3 pour la tâche C.
Entrez 4 pour la tâche D.
Entrez 5 pour sortir du programme.
1
Exécution de la tâche correspondant au choix 1

Entrez 1 pour la tâche A.
Entrez 2 pour la tâche B.
Entrez 3 pour la tâche C.
Entrez 4 pour la tâche D.
Entrez 5 pour sortir du programme.
6
Choix incorrect, essayez de nouveau.

Entrez 1 pour la tâche A.
Entrez 2 pour la tâche B.
Entrez 3 pour la tâche C.
Entrez 4 pour la tâche D.
Entrez 5 pour sortir du programme.
5
Sortie du programme ...
```

Analyse

Ce programme contient une fonction `menu()` qui est appelée à la ligne 17 et définie des lignes 58 à 71. Comme son nom l'indique, cette fonction propose un menu à l'utilisateur, et renvoie le choix de celui-ci au programme appelant. La fonction `main()` contient plusieurs instructions `if` imbriquées qui contrôlent l'exécution du programme en testant la valeur reçue. Ce programme ne fait rien d'autre qu'afficher des messages sur l'écran, mais il aurait pu appeler une fonction pour chaque option présentée dans le menu.

L'instruction *switch*

L'instruction `switch` est l'instruction de contrôle la plus souple du langage C. Elle permet à votre programme d'exécuter différentes instructions en fonction d'une expression qui pourra avoir plus de deux valeurs. Les instructions de contrôle précédentes, comme `if`, ne pouvaient évaluer que deux valeurs d'une expression : vrai ou faux. Cela obligerait, dans le cas d'un test de plus de deux valeurs, à utiliser des instructions imbriquées comme dans notre exemple du Listing 13.4. L'instruction `switch` résout ce problème de façon plus simple.

La syntaxe de `switch` est la suivante :

```
switch (expression)
{
    case modele_1: instruction(s);
    case modele_2: instruction(s);
...
    case modele_n: instruction(s);
    default: instruction(s);
}
```

`expression` représente une expression qui peut être évaluée avec une valeur entière de type `long`, `int` ou `char`. L'instruction `switch` évalue cette expression, compare la valeur obtenue avec les modèles fournis dans chaque instruction `case`, puis :

- Si l'expression correspond à un des modèles énoncés, l'instruction qui suit l'instruction `case` correspondante est exécutée.
- Si l'expression ne correspond à aucun modèle, c'est l'instruction située après l'instruction `default` qui est exécutée.
- Si l'expression ne correspond à aucun modèle, et que l'instruction `switch` ne contient pas l'instruction `default`, l'exécution se poursuit avec l'instruction qui suit l'accolade de fin de `switch`.

Le Listing 13.5 présente un programme simple, qui affiche un message en fonction du choix de l'utilisateur.

Listing 13.5 : Utilisation de l'instruction switch

```
/* Utilisation de l'instruction switch. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int reponse;

    puts("Entrez un nombre entre 1 et 5, ou 0 pour sortir : ");
    scanf("%d", &reponse);

    switch (reponse)
    {
    case 1:
      puts("Vous avez tapé 1.");
    case 2:
      puts("Vous avez tapé 2.");
    case 3:
      puts("Vous avez tapé 3.");
    case 4:
      puts("Vous avez tapé 4.");
    case 5:
      puts("Vous avez tapé 5.");
    default:
      puts("choix incorrect, essayez de nouveau.");
    }

    exit(EXIT_SUCCESS);
}
```

```
Entrez un nombre entre 1 et 5, ou 0 pour sortir :
2
Vous avez tapé 2.
Vous avez tapé 3.
Vous avez tapé 4.
Vous avez tapé 5.
Choix incorrect, essayez de nouveau.
```

Analyse

Ce résultat n'est pas satisfaisant. L'instruction `switch` a trouvé le modèle, mais elle a exécuté toutes les instructions suivantes (pas seulement celle qui est associée au modèle). En fait, `switch` réalise simplement un `goto` vers le modèle correspondant à l'expression. Pour n'exécuter que les instructions associées à ce modèle, vous devez inclure une instruction `break`. Dans le Listing 13.6, le programme précédent est revu et corrigé en ce sens.

Listing 13.6 : Association des instructions switch et break

```
/* Exemple d'utilisation correcte de l'instruction switch. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int reponse;

    puts("\nEntrez un nombre entre 1 et 5, ou 0 pour sortir : ");
    scanf("%d", &reponse);

    switch (reponse)
    {
    case 1:
     {
      puts("Vous avez choisi 1.\n");
      break;
     }
    case 2:
     {
      puts("Vous avez choisi 2.\n");
      break;
     }
    case 3:
     {
      puts("Vous avez choisi 3.\n");
      break;
     }
    case 4:
     {
      puts("Vous avez choisi 4.\n");
      break;
     }
    case 5:
     {
      puts("Vous avez choisi 5.\n");
      break;
     }
      default:
      puts("Choix incorrect, essayez de nouveau.\n");
     }
      /* fin du switch */
    exit(EXIT_SUCCESS);
}
```

```
list13_6
Entrez un nombre entre 1 et 5, ou 0 pour sortir :
1
Vous avez choisi 1.

list13_6
Entrez un nombre entre 1 et 5 :
6
Choix incorrect, essayez de nouveau.
```

Analyse

Compilez et exécutez cette version du programme ; le résultat obtenu est correct.

L'implémentation d'un menu comme celui du Listing 13.6 est une des utilisations les plus courantes de `switch`. Le code ainsi obtenu est bien meilleur que celui des instructions `if` imbriquées du Listing 13.4. Le programme du Listing 13.7 remplace les instructions `if` du Listing 13.4 par des instructions `switch`.

Listing 13.7 : Réalisation d'un menu système avec l'instruction switch

```
/* Utilisation d'une boucle infinie avec l'instruction switch */
/* pour implémenter un menu système. */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int menu(void);

int main()
{
  int choix;

  while (1)
  {
/* Branchement effectué en fonction du choix de l'utilisateur. */

    switch(choix=menu())
    {
      case 1:

      {
        puts("\nExécution du choix 1.");
        sleep(5);
        break;
      }
      case 2:
      {
        puts("\nExécution du choix 2.");
        sleep(5);
        break;
      }
      case 3:
      {
        puts("\nExécution du choix 3.");
        sleep(5);
        break;
      }
      case 4:
      {
        puts("\nExécution du choix 4.");
```

```
        sleep(5);
        break;
      }
      case 5:     /* Sortie du programme. */
      {
        puts("\nSortie du programme...\n");
        sleep(5);
        exit(0);
      }
      default:
      {
        puts("\nChoix incorrect, essayez de nouveau.");
        sleep(5);
      }
    }  /* Fin du switch   */
  }  /* Fin du while   */
  exit(EXIT_SUCCESS);
}

/* Affichage du menu et lecture du choix de l'utilisateur. */
int menu(void)
{
    int reponse;

    puts("\nEntrez 1 pour exécuter la tâche A.");
    puts("Entrez 2 pour exécuter la tâche B.");
    puts("Entrez 3 pour exécuter la tâche C");
    puts("Entrez 4 pour exécuter la tâche D.");
    puts("Entrez 5 pour sortir du programme.");

    scanf("%d", &reponse);

    return reponse;
}
```

```
Entrez 1 pour exécuter la tâche A.
Entrez 2 pour exécuter la tâche B.
Entrez 3 pour exécuter la tâche C.
Entrez 4 pour exécuter la tâche D.
Entrez 5 pour sortir du programme.
1
Exécution du choix 1.

Entrez 1 pour exécuter la tâche A.
Entrez 2 pour exécuter la tâche B.
Entrez 3 pour exécuter la tâche C.
Entrez 4 pour exécuter la tâche D.
Entrez 5 pour sortir du programme.
6

Choix incorrect, essayez de nouveau.

Entrez 1 pour exécuter la tâche A.
Entrez 2 pour exécuter la tâche B.
```

```
Entrez 3 pour exécuter la tâche C.
Entrez 4 pour exécuter la tâche D.
Entrez 5 pour sortir du programme.
5

Sortie du programme...
```

Analyse

Cette version du programme utilise une nouvelle fonction de bibliothèque : `exit()` (ligne 48). Cette fonction a été associée avec `case 5`, car l'instruction `break` ne peut être utilisée comme dans le Listing 13.4. En effet, `break` ne pourrait provoquer que la sortie de l'instruction `switch`, pas celle de la boucle infinie `while`. La fonction `exit()` arrête l'exécution du programme.

Dans certains cas, l'exécution "groupée" d'un sous-ensemble de l'instruction `switch` est nécessaire. Pour, par exemple, que le même bloc de code corresponde à plusieurs valeurs différentes, il faut supprimer les instructions `break` correspondantes. Le programme du Listing 13.8 vous en donne un exemple.

Listing 13.8 : Utilisation de l'instruction switch

```
/* Autre exemple d'utilisation de switch. */

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int reponse;

 while (1)
 {
   puts("\nEntrez une valeur entre 1 et 10, ou 0 pour sortir:  ");
   scanf("%d", &reponse);

   switch (reponse)
 {
   case 0:
   exit(EXIT_SUCCESS));
   case 1:
   case 2:
   case 3:
   case 4:
   case 5:
   {
   puts("Vous avez tapé un chiffre entre 1 et 5.\n");
   break;
   }
   case 6:
```

```
29:     case 7:
30:     case 8:
31:     case 9:
32:     case 10:
33:     {
34:     puts("Vous avez tapé un chiffre entre 6 et 10.\n");
35:     break;
36:     }
37:     default:
38:     puts("On a dit entre 1 et 10, s'il vous plait !\n");
39:       }  /* Fin du switch   */
40:      }   /* Fin du while     */
41:     exit(EXIT_SUCCESS);
42:  }
```

```
Entrez une valeur entre 1 et 10, ou 0 pour sortir :
11
On a dit entre 1 et 10, s'il vous plait !

Entrez une valeur entre 1 et 10, ou 0 pour sortir :
1
Vous avez tapé un chiffre entre 1 et 5.

Entrez une valeur entre 1 et 10, ou 0 pour sortir :
6
Vous avez tapé un chiffre entre 6 et 10.

Entrez une valeur entre 1 et 10, ou 0 pour sortir :
0
```

Analyse

Ce programme lit une valeur entrée au clavier pour savoir si elle se situe entre 1 et 5, entre 6 et 10, ou en dehors de l'intervalle 1-10. Si cette valeur est 0, la ligne 18 appelle la fonction `exit()` pour terminer le programme.

Syntaxe de l'instruction *switch*

```
switch (expression)
{
    case modele_1: instruction(s);
    case modele_2: instruction(s);
...
    case modele_n: instruction(s);
    default: instruction(s);
}
```

L'instruction `switch()` permet de réaliser plusieurs branchements à partir d'une seule expression. Elle est plus simple et plus efficace que l'emploi de plusieurs niveaux de `if`. L'instruction `switch` évalue l'expression pour se positionner sur l'instruction `case` dont le modèle correspond au résultat. Dans le cas où l'expression ne correspond à aucun

modèle, l'instruction `default` est exécutée. Si cette instruction n'existe pas, l'exécution se poursuit après la fin de l'instruction `switch`.

L'exécution se fait ligne par ligne, jusqu'à ce qu'une instruction `break` soit rencontrée. Le contrôle est alors transféré à la fin de l'instruction `switch`.

Exemple 1

```
switch (lettre)
{
    case 'A':
    case 'a':
      printf("vous avez tapé A");
      break;
    case 'B':
    case 'b':
      printf("vous avez tapé B");
      break;
    ...
    ...
    default:
      printf("je n'ai pas de réponse pour %c", lettre);
}
```

Exemple 2

```
switch (nombre)
{
    case 0 :    puts ("Votre nombre est 0 ou moins.");
    case 1 :    puts ("Votre nombre est 1 ou moins.");
    case 2 :    puts ("Votre nombre est 2 ou moins.");
    ...
    case 99 :   puts ("Votre nombre est 99 ou moins.");
      break;
    default :   puts ("Votre nombre est plus grand que 99.");
}
```

Les premières instructions `case` de cet exemple ne contiennent pas de `break`. L'exécution va donc afficher tous les messages de `case`, à partir du nombre saisi au clavier jusqu'au nombre 99 qui précède l'instruction `break`.



À ne pas faire

Oublier d'inclure les instructions `break` dont votre instruction `switch` a besoin.

À faire

Inclure la ligne `default` dans votre instruction `switch` même si vous pensez avoir prévu tous les cas de figure avec les différents `case`.

Utiliser une instruction `switch` plutôt qu'une instruction `if` si plus de deux conditions s'appliquent sur une même variable.

Aligner vos instructions `case` pour en faciliter la lecture.

Sortir du programme

Un programme C se termine naturellement quand l'exécution atteint l'accolade de fin de la fonction `main()`. Il est possible, toutefois, d'arrêter l'exécution à tout moment avec la fonction `exit()`, et de définir une ou plusieurs fonctions qui devront s'exécuter automatiquement à la fin du programme.

La fonction *exit()*

La fonction `exit()` interrompt l'exécution du programme et redonne le contrôle au système d'exploitation. Cette fonction possède un argument unique de type `int`, qui lui permet de transmettre au système d'exploitation une valeur indiquant le succès ou l'échec de l'exécution du programme. Cette fonction a la syntaxe suivante :

```
exit(status);
```

Si la valeur de `status` est 0, cela indique une fin normale du programme. Une valeur de 1 indique, au contraire, qu'une erreur s'est produite en cours d'exécution. En général, on ignore cette valeur. Consultez la documentation de votre système d'exploitation si vous voulez contrôler la valeur de retour d'un de vos programmes.

Le fichier en-tête stdlib.h doit être inclus pour que le programme puisse utiliser la fonction `exit()`. Ce fichier définit les deux constantes symboliques qui sont les arguments de cette fonction :

```
#define EXIT_SUCCESS   0
#define EXIT_FAILURE   1
```

Ces deux constantes permettent de sortir du programme. Avec une valeur de retour égale à 0, appelez `exit(EXIT SUCCESS)`. Avec une valeur de retour égale à 1, appelez `exit(EXIT  FAILURE)`.



À faire

Utiliser la commande `exit()` pour sortir du programme si une erreur se produit.

Transmettre des valeurs significatives à la fonction `exit()`.

Introduction de commandes système dans un programme

La bibliothèque standard de C contient une fonction, `system()`, qui permet d'introduire des commandes système dans vos programmes. Vous pourrez, par exemple, consulter la liste des répertoires d'un disque, ou formater un disque sans sortir du programme. Cette fonction est disponible avec le fichier en-tête stdlib.h. La syntaxe à respecter est la suivante :

```
system(commande);
```

L'argument `commande` peut être une constante chaîne de caractères, ou un pointeur vers une chaîne. Voici, par exemple, comment vous pourriez obtenir la liste d'un répertoire de Windows :

```
system("dir");
```

ou

```
char *commande = "dir";
system(commande);
```

Sur Linux et les systèmes Unix, remplacez "dir" par "ls".

À la fin de l'exécution de la commande système, le contrôle est rendu à l'instruction du programme qui suit l'appel de la fonction `system()`. Si la commande que vous transmettez est incorrecte, vous recevez un message `bad command` ou `file name error` ou un message de ce type avant de revenir dans le programme. Le Listing 13.9 vous donne un exemple d'utilisation de `system()`.

Listing 13.9 : Utilisation de la fonction system() pour exécuter des commandes système

```
/* Illustration de la fonction system(). */
#include <stdio.h>
#include <stdlib.h>

int main()
{
/* Déclaration d'une mémoire tampon pour y ranger les données */
/* entrées. */
     char input[40];

    while (1)
    {
       /* lecture de la commande utilisateur. */
       puts("\nEntrez une commande système, ou une ligne blanche \
pour sortir");
```

```
16:         lire_clavier(input, sizeof(input));
17:
18:         /* Sortie en cas de ligne blanche. */
19:
20:         if (input[0] == '\0')
21:            exit(EXIT_SUCCESS);
22:
23:         /* Exécution de la commande. */
24:
25:         system(input);
26:      }
27:      exit(EXIT_SUCCESS);
28:  }
```



```
Entrez une commande système, ou une ligne blanche pour sortir
dir *.bak

Volume in drive E is BRAD_VOL_B
Directory of E:\BOOK\LISTINGS
LIST1414 BAK    1416 05-22-97    5:18p
1 file(s)       1416 bytes
240068096 bytes free

Entrez une commande DOS, ou une ligne blanche pour sortir
```



`dir *.bak` est une commande DOS/Windows qui demande au système d'afficher tous les fichiers du répertoire courant ayant l'extension `bak`. Dans le cas d'une machine UNIX, vous devrez taper `ls *.bak` pour obtenir le même résultat.

Analyse

Ce programme lit les commandes système saisies par l'utilisateur en utilisant une boucle `while` (lignes 11 à 26). Si l'utilisateur n'entre pas de commande, la fonction `exit()` est appelée en ligne 21. Sinon, la ligne 25 appelle la fonction `system()` en lui transmettant la commande saisie par l'utilisateur. Bien sûr, ce programme ne donnera pas les mêmes résultats si vous le faites tourner sur votre système.

Vous pouvez transmettre à `system()` d'autres commandes que les simples `dir` ou `format`. Vous pouvez lui indiquer le nom de tout fichier exécutable (binaire, script...). Par exemple, si l'argument transmis est list13_8, vous allez exécuter le programme list13_8 avant de rendre le contrôle à l'instruction qui suit l'appel de `system()`.

Les seules restrictions concernant `system()` sont liées à la mémoire. Quand la fonction `system()` s'exécute, le programme qui l'a appelée reste chargé dans la mémoire RAM de votre ordinateur. Des copies de la commande système et du programme dont le nom a été transmis sont aussi chargées en mémoire. Si la mémoire est insuffisante, vous recevrez un message d'erreur.

Résumé

Le sujet traité dans ce chapitre est le contrôle du programme. Vous savez pourquoi il ne faut pas utiliser l'instruction `goto` (sauf quelques exceptions). Vous avez appris que les instructions `break` et `continue` permettent de contrôler l'exécution des boucles, et qu'elles vous seront très utiles avec les boucles infinies.

Vous savez comment sortir du programme avec la fonction `exit()`. Enfin, vous avez vu de quelle façon `system()` permet d'exécuter des commandes système depuis votre programme.

Q & R

Q Faut-il utiliser une instruction `switch` plutôt qu'une boucle imbriquée ?

R Si la comparaison se fait sur une variable qui peut avoir plus de deux valeurs, l'instruction `switch` sera presque toujours la meilleure solution. Le code sera aussi plus facile à lire. Si vous testez une condition du type vrai/faux, utilisez `if`.

Q Pourquoi faut-il éviter l'instruction `goto` ?

R L'instruction `goto` est attrayante, mais elle peut vous causer plus de problèmes qu'elle n'en résout. C'est une instruction non structurée qui vous branche à un autre emplacement du programme. Beaucoup de débogueurs (logiciels qui permettent de chercher des erreurs dans un programme) sont incapables d'interroger une instruction `goto` correctement. Cette instruction peut transformer votre code en code "spaghetti", c'est-à-dire en code dans lequel on se déplace de façon anarchique.

Q La fonction `system()` est-elle une bonne solution pour exécuter des commandes système ?

R La fonction `system()` est une solution simple pour des opérations telles que la réalisation d'une liste de fichiers dans un répertoire. Il faut toutefois être conscient du fait que les commandes système sont spécifiques à chaque système d'exploitation. Si votre programme utilise `system()`, il est probable qu'il ne sera plus portable. Ce problème de portabilité n'existe pas dans le cas où `system()` exécute un autre programme. Il existe par ailleurs quelques fonctions comme `execv()` ou `execl()` à combiner avec `fork()` pour obtenir des résultats plus complexes. Cela sort du cadre de ce livre.

Atelier

Cet atelier contient un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Quand faut-il utiliser une instruction `goto` dans un programme ?
2. Quelle est la différence entre une instruction `break` et une instruction `continue` ?
3. Qu'est-ce qu'une boucle infinie, et comment peut-on en créer une ?
4. Quels sont les deux événements qui provoquent la fin de l'exécution d'un programme ?
5. Quels sont les types de variables qu'une instruction `switch` peut évaluer ?
6. À quoi sert l'instruction `default` ?
7. À quoi sert la fonction `exit()` ?
8. À quoi sert la fonction `system()` ?

Exercices

1. Écrivez l'instruction qui provoque le démarrage immédiat de l'itération suivante d'une boucle.
2. Écrivez les instructions qui arrêtent un processus de bouclage pour donner le contrôle à l'instruction qui suit la fin de la boucle.
3. Écrivez la ligne de code qui affichera la liste des fichiers du répertoire courant .
4. **CHERCHEZ L'ERREUR :**

```
switch(reponse)
{
    case 'Y' : printf("Vous avez répondu oui");
      break;
    case 'N' : printf("Vous avez répondu non");
}
```

5. **CHERCHEZ L'ERREUR :**

```
switch (reponse)
{
    default :
      printf("Vous n'avez choisi ni 1 ni 2.");
    case 1 :
      printf("Vous avez répondu 1");
      break;
    case 2 :
      printf("vous avez répondu 2");
break;
}
```

6. Recodez l'exercice 5 en utilisant des instructions `if`.

7. Écrivez une boucle infinie `do while`.

   Les deux exercices qui suivent ayant une multitude de solutions, les réponses ne sont pas fournies en Annexe F.

8. TRAVAIL PERSONNEL : Écrivez un programme qui fonctionne comme une calculatrice. Il devra permettre de faire des additions, des soustractions, des multiplications et des divisions.

9. TRAVAIL PERSONNEL : Écrivez un programme qui possède un menu avec cinq options différentes. La cinquième permettra de sortir du programme. Les quatre autres permettront d'exécuter une commande système à l'aide de la fonction `system()`.

14

Travailler avec l’écran et le clavier

Les programmes effectuent presque tous des entrées/sorties. Vous avez déjà appris à réaliser les entrées/sorties de base. Aujourd’hui, vous allez étudier :

- L’utilisation des flots dans les entrées/sorties
- Les différentes façons de lire des données en provenance du clavier
- Les méthodes pour afficher du texte et des données numériques à l’écran
- La redirection des entrées et des sorties d’un programme

Les flots du C

Avant d'étudier dans le détail les entrées/sorties, vous devez savoir ce que sont les *flots*. Toutes les entrées/sorties du langage C se font avec les flots, quelle que soit leur origine ou destination. Vous constaterez que cette méthode standard de manipulation des entrées/sorties présente des avantages pour le programmeur. Il est bien sûr essentiel, de bien comprendre ce que représentent les flots et leur fonctionnement.

Définition des entrées/sorties

Vous savez maintenant qu'un programme en cours d'exécution range ses données dans la mémoire RAM. Ces données représentent les variables, les structures et les tableaux qui sont déclarés dans le programme. Voici l'origine de ces données et ce que le programme peut en faire :

- Les données sont issues d'un emplacement externe au programme. Les données qui sont transférées en mémoire RAM pour que le programme puisse y accéder, sont appelées les *entrées*. Les entrées proviennent le plus souvent du clavier ou de fichiers sur disque.
- Les données peuvent aussi être envoyées quelque part en dehors du programme : on les appelle alors les *sorties*. Les destinations les plus fréquentes sont l'écran, la sortie d'erreur et les fichiers disque.

Les sources d'entrées et les destinations de sorties sont appelées *unités*. Le clavier et l'écran, par exemple, sont des unités. Certaines unités (le clavier) sont exclusivement réservées aux entrées, d'autres (l'écran) servent aux sorties. D'autres encore (les fichiers disque) permettent de faire des entrées et des sorties.

Quelle que soit l'unité, le langage C effectue ses opérations d'entrées/sorties par l'intermédiaire des flots.

Définition d'un flot

Un *flot* est une séquence de caractères ou, plus exactement, une séquence d'octets de données. Une séquence d'octets qui arrive dans le programme est un flot d'entrées. Une séquence d'octets qui sort du programme est un flot de sorties. Le principal avantage des flots est que la programmation des entrées/sorties est indépendante des unités. Les programmeurs n'ont pas à créer de fonctions particulières d'entrées/sorties pour chaque unité. Le programme "voit" ses entrées/sorties comme un flot continu d'octets, quelle que soit la source ou la destination de ces données.

Chaque flot est connecté à un fichier. Le terme de fichier, ici, ne représente pas un fichier disque. Il s'agit plutôt d'une étape intermédiaire entre le flot avec lequel votre programme

travaille, et l'unité physique utilisée pour l'entrée ou la sortie. Ces fichiers ne sont d'aucune utilité au programmeur C débutant puisque le détail des interactions entre les flots, les fichiers et les unités est pris en charge par les fonctions de la bibliothèque du C et le système d'exploitation.

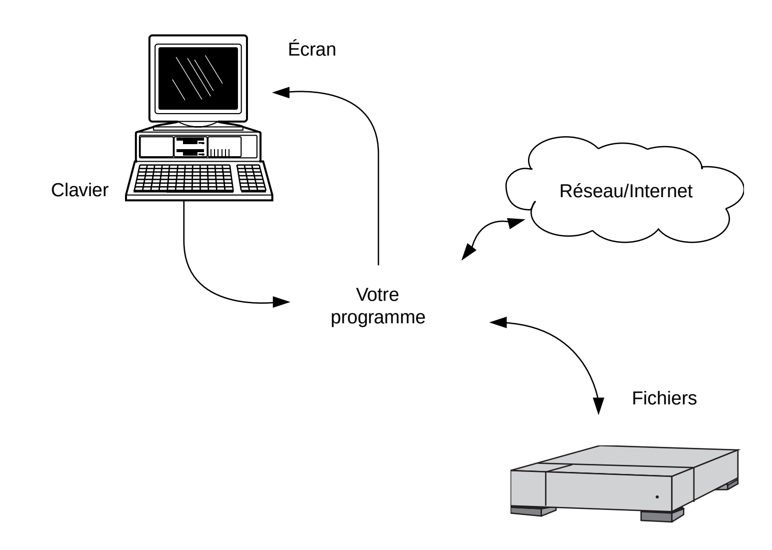


Figure 14.1
Les entrées/sorties constituent le lien entre votre programme et de nombreuses unités externes ou périphériques.

Flots de texte versus flots binaires

Il existe deux modes pour les flots de C : le mode texte ou le mode binaire. Un flot *texte* est constitué exclusivement de caractères, comme par exemple un message que l'on envoie sur l'écran. Le flot texte est divisé en lignes pouvant contenir 255 caractères chacune, qui se terminent par le caractère de retour à la ligne. Certains caractères de ce flot ont une signification particulière. Ce chapitre est consacré aux flots texte.

Un flot *binaire* peut contenir toute sorte de données, y compris les données texte. Les octets d'un flot binaire ne sont pas interprétés. Ils sont lus et écrits tels quels. Les flots binaires sont utilisés avec les fichiers disque étudiés au Chapitre 16.

Certains systèmes d'exploitation dont Linux ne font pas la différence entre flot texte et flot binaire : tout est considéré comme flot binaire.

Les flots prédéfinis

La norme ANSI contient trois flots prédéfinis appelés *fichiers entrées/sorties standard*. Ces flots sont ouverts automatiquement au début de l'exécution d'un programme C, et fermés à la fin du programme. Le Tableau 14.1 contient la liste des flots standard avec les

unités qui leur sont normalement connectées. Ces flots standard appartiennent au mode texte.

Tableau 14.1 : Les cinq flots standard de C

Nom	*Flot*	*Unité*
`stdin`	Entrée standard	Clavier
`stdout`	Sortie standard	Écran

Chaque fois que vous avez utilisé les fonctions `printf()` ou `puts()` pour afficher un texte à l'écran, vous avez utilisé le flot `stdout`. De la même façon, lorsque vous lisez les données entrées au clavier avec `scanf()`, vous utilisez le flot `stdin`.

Les fonctions d'entrées/sorties

La bibliothèque standard du C possède toute une variété de fonctions destinées aux entrées/sorties. Ces fonctions sont divisées en deux catégories : la première utilise toujours des flots standards, la seconde demande au programmeur de spécifier le flot. Le Tableau 14.2 donne une liste partielle de ces fonctions.

Tableau 14.2 : Les fonctions entrées/sorties de la bibliothèque standard

Utilise un flot standard	*Exige un nom de flot*	*Action*
`printf()`	`fprintf()`	Sortie formatée
`vprintf()`	`vfprintf()`	Sortie formatée avec une liste d'arguments
`puts()`	`fputs()`	Sortie chaîne
`putchar()`	`putc()`, `fputc()`	Sortie caractère
`scanf()`	`fscanf()`	Entrée formatée
`gets()` (à proscrire)	`fgets()`	Entrée chaîne
`getchar()`	`getc()`, `fgetc()`	Entrée caractère
`perror()`	–	Sortie chaîne pour `stderr`

L'utilisation de toutes ces fonctions requiert le fichier en-tête stdlib.h. Celui-ci devra aussi être inclus pour la fonction `perror()`, et le fichier stdargs.h est nécessaire pour les fonctions `vprintf()` et `vfprintf()`. Sur les systèmes UNIX, `vprintf()` et `vfprintf()`

pourront éventuellement nécessiter varargs.h. La documentation relative à la bibliothèque de votre système vous indiquera si des fichiers en-tête supplémentaires sont requis.

Exemple

Le Listing 14.1 propose un programme court qui démontre l'équivalence des flots.

Listing 14.1 : Équivalence des flots

```
 1:  /* Démonstration de l'équivalence des flots d'entrées et de sorties. */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:  int main()
 5:  {
 6:      char buffer[256];
 7:
 8:  /* Lecture d'une ligne, puis affichage immédiat de cette ligne.*/
 9:      lire_clavier(buffer, sizeof(buffer));
10:      puts(buffer);
11:
12:      exit(EXIT_SUCCESS);
13:  }
```

La fonction `lire clavier()` de la ligne 9 permet de lire une ligne de texte à partir du clavier. Cette fonction place la chaîne lue là où pointe `buffer`. Cette chaîne peut donc être utilisée comme argument pour la fonction `puts()` qui l'affiche à l'écran.



À faire

Profiter des avantages offerts par les flots d'entrées/sorties standards de C.

À ne pas faire

Transformer ou renommer les flots standards si ce n'est pas nécessaire.

Utiliser un flot d'entrée comme `stdin` pour une fonction comme `fprintf()` qui émet une "sortie".

Les entrées au clavier

Beaucoup de programmes C ont besoin de recevoir des informations en provenance du clavier (donc à partir de `stdin`). Les fonctions d'entrées sont divisées en trois catégories : les entrées caractère, les entrées ligne et les entrées formatées.

Entrées caractère

Les fonctions d'entrées caractère lisent les données du flot, caractère par caractère. Quand elles sont appelées, chacune de ces fonctions renvoie le caractère suivant du flot, ou `EOF` si on a atteint la fin du fichier ou si une erreur se produit. `EOF` est une constante symbolique définie dans stdio.h avec la valeur –1. Les fonctions d'entrées caractère présentent quelques différences concernant l'utilisation de la mémoire tampon et l'écho généré :

- Certaines fonctions d'entrées caractère utilisent la mémoire tampon. Cela signifie que le système d'exploitation stocke tous les caractères dans une mémoire temporaire, jusqu'à ce que vous appuyiez sur la touche Entrée. Les caractères sont envoyés dans le flot `stdin`. D'autres fonctions n'utilisent pas cette mémoire tampon, et les caractères sont alors envoyés un par un dans le flot `stdin`.
- Certaines fonctions d'entrées recopient automatiquement chaque caractère reçu dans le flot `stdout`. Les autres se contentent d'envoyer le caractère dans `stdin`. `stdout` représente l'écran, c'est donc sur l'écran que cet "écho" est généré.

La fonction *getchar()*

La fonction `getchar()` permet d'obtenir le caractère suivant du flot `stdin`. Elle utilise la mémoire tampon et recopie les caractères lus sur l'écran. Voici la syntaxe de sa déclaration :

```
int getchar(void);
```

Le Listing 14.2 illustre l'utilisation de `getchar()`. Le rôle de `putchar()`, qui sera expliqué plus loin dans ce chapitre, est d'afficher un simple caractère à l'écran.

Listing 14.2 : La fonction getchar()

```
/* Illustration de la fonction getchar(). */
#include <stdio.h>
#include <stdlib.h>

main()
{
    int ch;

    while ((ch = getchar()) != '\n')
        putchar(ch);
    exit(EXIT_FAILURE);

}
```



```
Voici ce que j'ai tapé.
Voici ce que j'ai tapé.
```

Analyse

La fonction `getchar()` est appelée à la ligne 9 et attend de recevoir un caractère de `stdin`. Cette fonction d'entrées utilisant la mémoire tampon, elle ne reçoit aucun caractère tant que vous n'avez pas enfoncé la touche Entrée. Dans le même temps, la valeur de chaque touche enfoncée est affichée à l'écran.

Quand vous appuyez sur Entrée, tous les caractères tapés, y compris le caractère de retour à la ligne, sont envoyés vers le flot `stdin` par le système d'exploitation. La fonction `getchar()` renvoie les caractères un par un, et les attribue un par un à la variable `ch`.

Chacun de ces caractères est comparé au caractère de retour à la ligne `"\n"`. S'il est différent de `\n`, le caractère est affiché à l'écran avec `putchar()`. Quand la fonction `getchar()` renvoie le caractère de retour à la ligne, la boucle `while` se termine.

La fonction `getchar()` peut être utilisée pour lire des lignes de texte, comme le montre le Listing 14.3. Vous apprendrez plus loin dans ce chapitre que d'autres fonctions conviennent mieux à ce type d'opération, en particulier la fonction `lire clavier()` que nous utilisons depuis le début de ce livre.

Listing 14.3 : Lecture d'une ligne de texte avec getchar()

```
/*    Utilisation de getchar() pour lire des chaînes de caractères.*/
#include <stdio.h>
#include <stdlib.h>

#define MAX 80

int main()
{
    char ch, buffer[MAX+1];
    int x = 0;

    while ((ch = getchar()) != '\n' && x < MAX)
        buffer[x++] = ch;

    buffer[x] = '\0';

    printf("%s\n", buffer);

    exit(EXIT_SUCCESS);
}
```



```
Cela est une chaîne
Cela est une chaîne
```

Analyse

Ce programme utilise `getchar()` comme celui du Listing 14.2. La boucle contient toutefois une condition supplémentaire. La boucle `while` accepte de recevoir des caractères de la part de `getchar()` jusqu'à ce qu'elle trouve un caractère de retour à la ligne, ou que le nombre de caractères lus soit de 80. Chaque caractère est stocké dans le tableau `buffer[]`. Quand tous les caractères ont été lus, la ligne 15 ajoute le caractère nul à la fin du tableau pour que la fonction `printf()` de la ligne 17 puisse afficher la chaîne.

La taille choisie pour le tableau `buffer` (ligne 9) est de `MAX+1`. Ce choix permettra de lire une chaîne de 80 caractères à laquelle on ajoutera le caractère nul (à ne pas oublier).

La fonction *getch()*

La fonction `getch()` permet d'obtenir le caractère suivant du flot `stdin`. Elle n'utilise pas la mémoire tampon et n'envoie pas d'écho vers l'écran. Cette fonction ne fait partie ni du standard ANSI/ISO ni d'un standard répandu tel que POSIX ou BSD, elle pourrait donc ne pas être disponible sur tous les systèmes, en particulier Unix et Linux. Elle pourrait de plus imposer l'inclusion de divers fichiers en-tête. La déclaration de cette fonction, qui se trouve généralement dans le fichier en-tête conio.h, a la forme suivante :

```
int getch(void);
```

`getch()` envoie chaque caractère lu au clavier vers `stdin` sans attendre que l'utilisateur appuie sur la touche Entrée. Elle ne recopie pas ces caractères dans `stdout`, vous ne les verrez donc pas apparaître à l'écran.



Le listing qui suit utilise `getch()`, qui n'appartient ni au standard ANSI/ISO ni à un standard répandu tel que POSIX ou BSD. Utilisez prudemment ce type de fonction, car elle pourrait ne pas être reconnue par certains compilateurs. Si vous obtenez des erreurs en compilant ce listing, votre compilateur en fait peut-être partie.

Listing 14.4 : La fonction getch()

```
/* Utilisation de la fonction getch(). */
/* code non ANSI  */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int main()
{
    int ch;

```

```
10:      while ((ch = getch()) != '\r')
11:          putchar(ch);
12:      exit(EXIT_SUCCESS);
13:  }
```



```
Test de la fonction getch()
```

Analyse

Quand ce programme s'exécute, `getch()` envoie immédiatement vers `stdin`, chaque caractère que vous avez tapé. Cette fonction n'envoyant pas d'écho vers la sortie standard, vous voyez apparaître vos caractères sur l'écran grâce à la fonction `putchar()`. Pour mieux comprendre le fonctionnement de `getch()`, ajoutez un point-virgule à la fin de la ligne 10 et supprimez la ligne 11 (`putchar()`). En exécutant de nouveau ce programme, vous constaterez que les caractères tapés ne sont plus affichés. La fonction `getch()` les récupère en effet sans les reproduire à l'écran. Vous savez que ces caractères ont bien été lus parce que le listing initial avait fait appel à `putchar()` pour les afficher.

Ce programme compare chaque caractère reçu avec le caractère `\r`, qui est le code correspondant au caractère "retour chariot". Quand vous appuyez sur la touche Entrée, le clavier envoie un caractère retour chariot (CR) vers `stdin`. Les fonctions d'entrées caractère qui utilisent la mémoire tampon convertissent automatiquement ce retour chariot en caractère de retour à la ligne. C'est pourquoi les programmes testent plutôt, dans ce cas, le caractère `\n` pour savoir si l'utilisateur a appuyé sur **Entrée**.

Le Listing 14.3 vous présente un exemple d'utilisation de `getch()` pour lire une ligne entière de texte. En exécutant ce programme, vous pourrez constater l'absence d'écho à l'écran avec `getch()`.

Listing 14.5 : Lecture d'une ligne de texte avec la fonction getch()

```
/* Utilisation de getch() pour lire des chaînes de caractères. */
/* Code non ISO/ANSI  */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 80

int main()
{
    char ch, buffer[MAX+1];
    int x = 0;

    while ((ch = getch()) != '\r' && x < MAX)
        buffer[x++] = ch;

```

Listing 14.5 : Lecture d'une ligne de texte avec la fonction getch() ***(suite)***

```
    buffer[x] = '\0';

    printf("%s", buffer);
    exit(EXIT_SUCCESS);
}

```



```
Cela est une chaîne
Cela est une chaîne
```

La fonction *getche()*

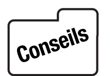
La seule différence entre `getch()` et `getche()` est que celle-ci envoie un écho des caractères vers l'écran. Modifiez le programme du Listing 14.4 en remplaçant `getch()` par `getche()`. Vous pourrez constater que chaque caractère tapé sera affiché deux fois sur l'écran : la première fois est due à l'écho de `getche()`, et la seconde, à la fonction `putchar()`.



`getche()` n'est pas une fonction standard ISO/ANSI ni d'aucun standard répandu comme POSIX ou BSD.

Les fonctions *getc()* et *fgetc()*

Ces deux fonctions d'entrées caractère n'utilisent pas automatiquement `stdin` ; le flot d'entrées doit être spécifié par le programme. Elles sont utilisées pour lire des caractères à partir des fichiers disque qui sont étudiés au Chapitre 16.



À faire

Distinguer les entrées simples des entrées qui sont envoyées en écho sur l'écran.

Distinguer les entrées qui utilisent la mémoire tampon de celles qui ne l'utilisent pas.

À ne pas faire

Utiliser des fonctions non ISO/ANSI pour obtenir un code portable.

Comment "rendre" un caractère avec *ungetc()*

Nous allons utiliser un exemple pour vous expliquer ce que signifie "rendre" un caractère. Imaginons que votre programme soit en train de lire les caractères du flot d'entrées, et que

la seule méthode pour détecter la fin de cette lecture soit de lire un caractère de trop. Par exemple, si vous lisez des chiffres, vous saurez qu'il n'y a plus de données à lire quand vous rencontrerez le premier caractère qui ne sera pas un chiffre. Celui-ci est le premier caractère de l'entrée suivante, mais il ne se trouve plus dans le flot. Vous pouvez l'y replacer, ou le "rendre", pour que la prochaine opération sur ce flot puisse lire ce caractère.

Pour cela, vous devez utiliser la fonction de la bibliothèque `ungetc()` dont la déclaration a la forme suivante :

```
int ungetc(int ch, FILE *fp);
```

L'argument `ch` est le caractère à renvoyer. L'argument `*fp` identifie le flot vers lequel le caractère doit être envoyé. La notation `FILE *fp` est utilisée pour les flots associés à des fichiers disque (voir Chapitre 16). Si le flot est le flot standard vous écrirez :

```
ungetc(ch, stdin);
```

Vous ne pouvez rendre qu'un seul caractère à un flot entre deux lectures, et ce caractère ne peut être le caractère `EOF`. Le programme du Listing 17.16 utilise cette fonction `ungetc()`.

Entrées ligne

Les fonctions d'entrées ligne lisent tous les caractères d'un flot d'entrées jusqu'au premier caractère de retour à la ligne. La bibliothèque standard contient deux fonctions d'entrées ligne : `gets()` et `fgets()`.

La fonction gets()

Mise en garde : cette fonction est dangereuse. Nous la présentons ici pour mieux vous montrer pourquoi elle est à proscrire.

La fonction `gets()` permet de lire une ligne complète dans `stdin` et de la stocker dans une chaîne de caractères. La déclaration de cette fonction a la syntaxe suivant :

```
char *gets(char *str);
```

L'argument de cette fonction est un pointeur vers une variable de type `char`, et la valeur renvoyée est un pointeur du même type. Elle lit les caractères de `stdin` jusqu'à ce qu'elle rencontre le caractère de retour à la ligne (`\n`) ou une fin de fichier. Le caractère de retour à la ligne est remplacé par le caractère nul, et la chaîne est stockée à l'adresse pointée par `str`.

La valeur renvoyée est un pointeur vers la chaîne de caractères lue (le même que `str`). Si `gets()` rencontre une erreur ou lit une fin de fichier avant de lire un caractère, le pointeur renvoyé sera un pointeur nul.

Avant d’appeler la fonction `gets()`, il faut allouer assez de mémoire pour stocker la chaîne. La fonction ne peut pas savoir si l’adresse pointée par `ptr` est un emplacement réservé ou non. Dans tous les cas, la chaîne sera stockée à cet endroit. Si l’espace n’a pas été réservé auparavant, les données qui s’y trouvaient sont écrasées. Mais surtout, si l’espace réservé était insuffisant, les données au-delà de la zone réservée sont également écrasées. Comme il est impossible de connaître la taille de l’espace mémoire à réserver, le programme n’est donc jamais à l’abri d’un utilisateur malveillant (autrement dit, un pirate) qui entrerait plus de caractères que le programmeur en a prévu et qui pourrait de cette façon agir sur votre ordinateur de façon indésirable.

Dans ces conditions, soyez-en prévenus : n’utilisez jamais gets(), mais fgets(), présentée plus loin et qui, utilisée comme ci-dessous, est équivalente à gets(), sans le problème de sécurité :

```
char buffer[80];

fgets(buffer, sizeof(buffer), stdin);
```

La fonction fgets()

Comme la fonction `gets()`, `fgets()` permet de lire une ligne de texte dans un flot d’entrées. Elle est plus souple, car elle autorise le programmeur à spécifier le flot d’entrées à utiliser et le nombre maximum de caractères à lire. Cette fonction est souvent utilisée pour lire du texte dans des fichiers disque (voir Chapitre 16). Voici la déclaration de `fgets()` :

```
char *fgets(char *str, int n, FILE *fp);
```

Le dernier paramètre `FILE *fp` représente le flot d’entrées. Contentez-vous aujourd’hui de le remplacer par `stdin`.

Le pointeur `str` indique où la chaîne est stockée, et l’argument `n` est le nombre maximum de caractères à lire. `fgets()` va lire les caractères du flot d’entrées jusqu’à ce qu’elle rencontre un caractère de fin de ligne ou de retour à la ligne, ou qu’elle ait lu $n - 1$ caractères. Le caractère de retour à la ligne est inclus dans la chaîne avant de la stocker. Les valeurs renvoyées par `fgets()` sont les mêmes que celles de la fonction `gets()`.

Pour tout dire, si vous définissez une ligne comme étant une suite de caractères terminée par un retour à la ligne, `fgets()` ne lit pas une simple ligne de texte. Elle ne lira qu’une partie d’une ligne contenant plus de $n - 1$ caractères. Avec `stdin`, l’exécution de `fgets()` se poursuivra jusqu’à ce que vous tapiez sur la touche **Entrée**, mais seuls $n - 1$ caractères seront stockés dans la chaîne. Le Listing 14.6 présente le fonctionnement de la fonction `fgets()`.

Listing 14.6 : La fonction fgets()

```
 1:  /* Exemple d'utilisation de la fonction fgets(). */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  #define MAXLONG 10
 6:
 7:  int main()
 8:  {
 9:     char buffer[MAXLONG];
10:
11:     puts("Entrez une ligne de texte à la fois, ou un blanc \
12:  pour sortir.");
13:     while (1)
14:     {
15:        fgets(buffer, MAXLONG, stdin);
16:
17:        if (buffer[0] == '\n')
18:           break;
19:
20:        puts(buffer);
21:     }
22:
23:     exit(EXIT_SUCCESS);
24:  }
```

```
Entrez une ligne de texte à la fois, ou un blanc pour sortir.
Les roses sont rouges
Les roses
sont rou
ges

Les violettes sont bleues
Les viole
ttes sont
bleues

Programmer en C
programme
r en C

Est bon pour vous !
Est bon p
our vous
!
```

La fonction `fgets()` apparaît en ligne 15. En exécutant ce programme, entrez des lignes d'une longueur inférieure, puis supérieure à `MAXLEN` et observez le résultat. Dans le cas d'une longueur supérieure, le premier appel de `fgets()` lit les `MAXLEN 1` premiers caractères, les autres étant stockés dans la mémoire tampon du clavier. Ils sont lus ensuite par un second appel de `fgets()` ou par toute autre fonction qui lit à partir de `stdin`. Le programme se termine lorsqu'il reçoit une ligne vide (lignes 17 et 18).

Les entrées formatées

Les fonctions d'entrées que nous avons étudiées jusqu'à présent prenaient simplement un ou plusieurs caractères dans un flot d'entrées pour les stocker quelque part en mémoire. Ces caractères n'étaient ni formatés, ni interprétés, et vous ne pouviez pas lire des variables numériques. Pour lire à partir du clavier la valeur 12.86, par exemple, et l'attribuer à une variable de type `float`, vous devrez utiliser les fonctions `scanf()` et `fscanf()`. Nous avons étudié ces fonctions au Chapitre 7.

Ces deux fonctions sont analogues. `scanf()` utilise toujours `stdin`, alors que `fscanf()` utilise le flot d'entrées spécifié par l'utilisateur. `fscanf()` est utilisée pour les entrées fichier qui sont traitées au Chapitre 16.

Les arguments de la fonction *scanf()*

La fonction `scanf()` prend un minimum de deux arguments. Le premier est la chaîne format qui utilise des caractères spéciaux pour indiquer à `scanf()` comment interpréter les entrées. Les arguments suivants représentent les adresses des variables dans lesquelles seront stockées les données d'entrées. Voici un exemple :

```
scanf("%d", &x);
```

Le premier argument "`%d`" est la chaîne format. Dans cet exemple, "`%d`" indique à `scanf()` de chercher une valeur entière signée. Le second argument utilise l'opérateur d'adresse (`&`) pour demander à `scanf()` d'attribuer la valeur de l'entrée à la variable `x`. Examinons en détail la chaîne format.

Voici les trois éléments que l'on peut introduire dans une chaîne format :

- Des blancs et des tabulations qui seront ignorés.
- Des caractères (sauf %) qui seront associés à tous les caractères lus (sauf les blancs).
- Une ou plusieurs conversions qui seront constituées du caractère % suivi d'un caractère particulier. En général, la chaîne format contient une conversion par variable.

Les conversions représentent la seule partie obligatoire de la chaîne format. Chacune d'entre elles commence par le caractère % suivi de composants obligatoires ou optionnels dans un certain ordre. La fonction `scanf()` applique les demandes de conversions de la chaîne format, dans l'ordre, aux champs du flot d'entrées. Un *champ d'entrées* est une séquence de caractères qui se termine au premier blanc rencontré ou quand la largeur de champ spécifiée est atteinte. Les composants d'un ordre de conversion sont les suivants :

- L'indicateur de suppression d'attribution (`*`) qui est optionnel et placé immédiatement après %. Ce caractère demande à `scanf()` de réaliser la conversion en cours, mais d'en ignorer le résultat (ne pas en attribuer la valeur à la variable).

- La largeur de champ qui est aussi optionnelle. Ce composant est un nombre décimal qui indique le nombre de caractères du champ d'entrées. En d'autres termes, la largeur de champ indique à `scanf()` le nombre de caractères qu'elle doit prendre en compte dans `stdin` pour la conversion en cours. Si la largeur de champ n'est pas indiquée, `scanf()` lira tous les caractères jusqu'au premier blanc.



Si vous lisez une chaîne de caractères (indicateur de type s), il est impératif (et malheureusement facultatif) que vous indiquiez une largeur de champ. Sinon, vous introduisez une faille de sécurité dans votre code, la même que celle décrite plus haut relative à `gets()`.

- Le composant suivant est toujours optionnel, il s'agit de l'attribut de précision. Celui-ci est le simple caractère `h`, `l` ou `L`. Il permet de changer la signification du type de variable qui le suit (voir plus loin).
- Le seul composant obligatoire d'une conversion (en dehors de %) est l'indicateur du type. Cet indicateur est représenté par un ou plusieurs caractères indiquant à `scanf()` comment il doit interpréter les données lues. Le Tableau 14.3 donne la liste de ces caractères avec leur explication. La colonne "Argument" donne le type de variable correspondant à l'indicateur de type de `scanf()`. Un indicateur de type `d`, par exemple, doit s'appliquer à une variable lue de type `int *` (un pointeur de variable de type `int`).

Tableau 14.3 : Liste des caractères utilisés pour les spécifications de conversion de scanf()

Type	*Argument*	*Signification*
`d`	`int *`	Entier décimal.
`i`	`int *`	Entier exprimé en base 10, 8 (premier chiffre 0) ou 16 (commence par 0X ou 0x).
`o`	`int *`	Entier exprimé en base 8 avec ou sans 0 devant.
`u`	`unsigned int *`	Entier décimal non signé.
`x`	`int *`	Entier hexadécimal avec ou sans 0X ou 0x devant.
`c`	`char *`	Les caractères sont lus et stockés séquentiellement à l'adresse mémoire indiquée par l'argument. Le caractère de fin \0 n'est pas ajouté. Sans argument de largeur de champ, la lecture se fait sur un seul caractère. Si cet argument est donné, la lecture se fait sur le nombre de caractères indiqué en incluant les blancs.
`s`	`char *`	La chaîne de caractères (sans blancs) est lue et stockée à l'adresse indiquée par l'argument en lui ajoutant le caractère de fin \0.

Tableau 14.3 : Liste des caractères utilisés pour les spécifications de conversion de scanf() *(suite)*

Type	*Argument*	*Signification*
`e,f,g`	`float *`	Nombre avec une virgule flottante. Ces nombres peuvent être lus en notation décimale ou scientifique.
`[...]`	`char *`	Chaîne de caractères. Seuls, les caractères entre crochets sont lus. La lecture est interrompue dès l'arrivée d'un caractère ne faisant pas partie de la liste, ou dès que le nombre de caractères atteint la largeur de champ, ou encore si la touche Entrée est enfoncée. Pour que le caractère `]` soit lu, il doit être placé en premier : `[]...]`. `\0` est ajouté en fin de chaîne.
`[^...]`	`char *`	Analogue à `[...]`. Tous les caractères sont acceptés à l'exception de ceux entre crochets.
`%`	`None`	Ce caractère n'est pas stocké, il est seulement lu comme le caractère %.

Avant de voir des exemples d'utilisation de la fonction `scanf()`, vous devez comprendre le rôle des attributs de précision du Tableau 14.4.

Tableau 14.4 : Les attributs de précision

Attribut de précision	*Signification*
`h`	Placé avant l'indicateur de type `d`, `i`, `o`, `u` ou `x`, l'attribut `h` indique que l'argument est un pointeur vers une variable de type `short` , plutôt que de type `int`. Sur un PC, `short` et `int` sont identiques ; `h` ne sera donc jamais utilisé.
`l`	Quand on le place avant l'indicateur de type `d`, `i`, `o`, `u` ou `x`, il indique que l'argument est un pointeur de type `long`. Placé avant `e`, `f` ou `g`, l'attribut `l` indique que l'argument est un pointeur de type `double`.
`L`	Placé avant l'indicateur de type `e`, `f` ou `g`, cet attribut indique que l'argument est un pointeur de type `long double`.

Le traitement des caractères parasites

La fonction `scanf()` utilise la mémoire tampon. Elle ne reçoit aucun caractère tant que l'utilisateur n'a pas appuyé sur la touche **Entrée**. La ligne entière de caractères est alors envoyée dans `stdin` pour être traitée par `scanf()`. L'exécution de `scanf()` se termine lorsqu'elle a traité le nombre de champs d'entrées correspondant aux conversions de sa chaîne format. Si des caractères subsistent dans `stdin` après la fin de l'exécution de `scanf()`, ils risquent de provoquer des erreurs. Examinons le fonctionnement de `scanf()` pour en comprendre la raison.

Quand `scanf()` est appelée alors que l'utilisateur vient d'entrer une ligne de texte, trois situations peuvent se présenter. Pour notre exemple, supposons que l'instruction `scanf("%d %d", &x, &y)` ait été exécutée. Elle s'attend à recevoir deux entiers décimaux :

- Premier cas, la ligne entrée par l'utilisateur correspond en tout point à la chaîne format. Il a pu, par exemple, saisir `12 14` puis **Entrée**. `scanf()` s'exécute normalement et le flot `stdin` ne contiendra plus de caractères.
- Deuxième cas, la ligne entrée par l'utilisateur ne contient pas assez d'éléments pour satisfaire la chaîne format. Il a pu saisir, par exemple, `12` puis **Entrée**. `scanf()` va alors attendre les données manquantes. L'exécution se poursuit après la réception de ces données et `stdin` ne contient plus de caractères.
- Troisième cas, la ligne entrée par l'utilisateur a plus d'éléments que n'en demande la chaîne format. L'utilisateur a tapé `12 14 16` puis **Entrée**, par exemple. `scanf()` va lire `12` et `14` avant de rendre le contrôle au programme appelant. Les deux caractères supplémentaires `1` et `6` vont rester en attente dans `stdin`.

Voici pourquoi cette troisième situation peut être à l'origine de problèmes. Ces deux caractères vont rester dans `stdin` aussi longtemps que le programme s'exécutera. Ils seront donc présents lors de la lecture suivante de `stdin` et ils seront les premiers caractères traités.

Pour éviter de telles erreurs, vous avez deux solutions. La première est que les utilisateurs de vos programmes ne se trompent jamais. Elle sera plutôt difficile à mettre en œuvre.

Une meilleure solution consiste à s'assurer qu'aucun caractère parasite ne subsiste dans le flot avant de demander des données à l'utilisateur. Vous pouvez, pour cela, appeler `fgets()` qui lira tous les caractères restant dans `stdin`, y compris le caractère de fin de ligne. Plutôt que d'appeler `fgets()` directement dans vos programmes, vous pouvez créer une fonction spécifique appelée `clear kb()` comme le montre le Listing 14.7.

Listing 14.7 : Suppression des caractères parasites de stdin

```
/* Nettoyage de stdin. */
#include <stdio.h>
#include <stdlib.h>

void clear_kb(void);

int main()
{
   int age;
   char nom[20];

   /* On demande l'âge de l'utilisateur. */
```

Listing 14.7 : Suppression des caractères parasites de stdin *(suite)*

```

   puts("Entrez votre âge : ");
   scanf("%d", &age);

   /* On retire de stdin les caractères parasites. */

   clear_kb();

   /* Lecture du nom de l'utilisateur. */

   puts("Entrez votre nom : ");
   scanf("%19s", nom);
   /* Affichage des donn,es. */

   printf("Vous avez %d ans.\n", age);
   printf("Vous vous appelez %s.\n", nom);

   exit(EXIT_SUCCESS);
}

void clear_kb(void)

/* On efface de stdin les caractères restants. */
{
   char junk[80];
   fgets(junk, sizeof(junk) stdin);
}
```

```
Entrez votre âge :
29 et pas un an de plus!
Entrez votre nom :
Bradley
Vous avez 29 ans.
Vous vous appelez Bradley.
```

Analyse

En exécutant le programme du Listing 14.7, entrez des caractères supplémentaires après votre âge. Assurez-vous que le programme les ignore et qu'il interprète correctement votre nom. Modifiez ensuite ce programme en supprimant l'appel de la fonction `clear kb()` et relancez-le. Les caractères parasites tapés après votre âge vont être attribués à votre nom.

Le traitement des caractères parasites avec *fflush()*

Il existe une autre méthode pour effacer les caractères superflus. La fonction `fflush()` fait disparaître les informations présentes dans un flot, y compris le flot d'entrée standard. Elle est généralement utilisée avec les fichiers sur disque (traités au Chapitre 16). Elle peut

cependant simplifier le Listing 14.7. Le Listing 14.8 l'utilise à la place de la fonction `clear kb()`, qui avait été créée dans le Listing 14.7.

Listing 14.8 : Nettoyage de stdin avec fflush ()

```
1:   /* Nettoyage de stdin avec fflush(). */
2:   #include <stdio.h>
3:   #include <stdlib.h>
4:
5:    int main()
6:    {
7:       int age;
8:       char name[20];
9:
10:     /* On demande l'âge de l'utilisateur. */
11:     puts("Entrez votre âge.");
12:     scanf("%d", &age);
13:
14:     /* On retire de stdin les caractères parasites. */
15:     fflush(stdin);
16:
17:     /* Lecture du nom de l'utilisateur. */
18:     puts("Entrez votre nom.");
19:     scanf("%19s", name);
20:
21:     /* Affichage des données. */
22:     printf("Vous avez %d ans.\n", age);
23:     printf("Vous vous appelez %s.\n", name);
24:
25:     exit(EXIT_SUCCESS);
26: }
```

L'exécution de ce programme donne le résultat :

```
Entrez votre âge.
29 et pas un an de plus!
Entrez votre nom.
Bradley
Vous avez 29 ans.
Vous vous appelez Bradley.
```

La fonction `fflush()` apparaît en ligne 15, et son prototype est le suivant :

```
int fflush( FILE *flot);
```

`flot` représente le flot à "nettoyer". Dans le Listing 14.8, c'est le flot d'entrée standard `stdin` qui a été transmis comme `flot`.

Exemples avec *scanf()*

La meilleure façon de se familiariser avec les opérations de la fonction `scanf()`, c'est de les tester. Le programme du Listing 14.10 vous présente quelques utilisations de cette fonction parmi les moins courantes. Compilez ce programme puis exécutez-le. Faites ensuite quelques tests en modifiant les chaînes format de `scanf()`.

Listing 14.9 : Exemples d'utilisation de scanf() pour les entrées au clavier

```
/* Exemple d'utilisations de scanf(). */

#include <stdio.h>
#include <stdlib.h>


int main()
{
   int i1, i2;
   long l1;

   double d1;
   char buf1[80], buf2[80];

/* Utilisation de l pour entrer des entiers de types long et
double. */
   puts("Entrez un entier et un nombre avec une virgule : ");
   scanf("%ld %lf", &l1, &d1);
   printf("\nVous avez tapé %ld et %lf.\n",l1, d1);
   puts("La chaîne format de scanf() a utilisé l'attribut l");
   puts("pour stocker vos données dans une variable de type");
   puts("long et une autre de type double.\n");

   fflush(stdin);

/* Utilisation de la largeur du champ pour couper les données
entrées. */
   puts("Entrez un entier de 5 chiffres (par exemple, 54321) :");
   scanf("%2d%3d", &i1, &i2);

   printf("\nVous avez tapé %d et %d.\n", i1, i2);
   puts("Notez comment l'indicateur de largeur de champ ");
   puts("de la chaîne format de scanf() a séparé votre valeur \
        en deux.\n");
   fflush(stdin);

/* Utilisation d'un espace exclu pour stocker une ligne */
/* entrée dans deux chaînes. */

   puts("Entrez vos nom et prénom séparés par un blanc : ");
   scanf("%[^ ]%80s", buf1, buf2);
   printf("\nVotre nom est %s\n", buf1);
   printf("Votre prénom est %s\n", buf2);
```

```
44:     puts("Notez comment le caractère [^ ] de la chaîne format de");
45:     puts("scanf(), en excluant le caractère blanc, a séparé \
46:           les données entrées.");
47:     exit(EXIT_SUCCESS);
48:  }
```

```
Entrez un entier et un nombre avec une virgule :
123 45.6789

Vous avez tapé 123 et 45.678900.
La chaîne format de scanf() a utilisé l'attribut l
pour stocker vos données dans une variable de type
long et une autre de type double.

Entrez un entier de 5 chiffres (par exemple 54321) :
54321

Vous avez tapé 54 et 321.
Notez comment l'indicateur de largeur de champ
de la chaîne format de scanf() a séparé votre valeur en deux.

Entrez vos nom et prénom séparés par un blanc :
Peter Aitken
Votre nom est Peter,
Votre prénom est Aitken
Notez comment le caractère [^ ] de la chaîne format de
scanf(), en excluant le caractère blanc, a séparé les données entrées.
```

Analyse

Le source de ce programme commence avec la définition de plusieurs variables (lignes 9 à 13) qui contiendront les données lues. Le programme demande ensuite à l'utilisateur d'entrer divers types de données. Les lignes 17 à 22 lisent et affichent un entier `long` et un autre de type `double`. La ligne 24 appelle la fonction `fflush()` pour effacer les caractères parasites du flot d'entrées. Les lignes 28 et 29 demandent ensuite un entier de 5 chiffres. La chaîne format contenant des indications de largeur de champ, cet entier est coupé en deux. La fonction `fflush()` est appelée de nouveau à la ligne 35 pour vider `stdin`. Le dernier exemple (lignes 40 à 47), utilise le caractère d'exclusion. En effet, "`%[^]%80s`" à la ligne 41 demande à `scanf()` de lire une chaîne de caractères et d'arrêter sa lecture au premier blanc rencontré. Cela a permis de séparer les deux mots entrés sur la même ligne.

La fonction `scanf()` peut être utilisée pour traiter la plupart de vos entrées, en particulier celles qui contiennent des nombres (les chaînes sont traitées plus facilement avec `fgets()`). Toutefois, il est souvent préférable d'écrire ses propres fonctions d'entrées. Le Chapitre 18 vous présentera quelques exemples de fonctions d'entrées utilisateur.

Conseils

À faire

Utiliser les caractères étendus dans vos programmes afin de conserver une bonne cohérence avec les autres programmes.

Utiliser `scanf()` plutôt que `fscanf()` si vous n'utilisez que le flot `stdin` et que vous lisez des nombres. Réservez `fgets()` pour les chaînes de caractères.

À ne pas faire

Oublier de contrôler la présence de caractères parasites dans le flot d'entrées.

Utiliser `gets()`.

Utiliser `scanf()` pour des chaînes de caractères sans préciser d'indicateur de taille.

Les sorties écran

Les fonctions de sorties écran sont divisées en trois catégories principales, sur le même principe que les fonctions d'entrées : les sorties caractère, les sorties ligne et les sorties formatées. Nous avons utilisé quelques-unes de ces fonctions dans les chapitres précédents.

Sorties caractère avec *putchar()*, *putc()* et *fputc()*

Les fonctions de sorties caractère de la bibliothèque C envoient un simple caractère dans un flot. La fonction `putchar()` envoie ses caractères vers `stdout` (généralement l'écran). Les fonctions `fputc()` et `putc()` envoient leurs sorties dans le flot indiqué dans la liste des arguments.

Utilisation de *putchar()*

La déclaration de `putchar()` se trouve dans le fichier en-tête stdio.h :

```
int putchar(int c);
```

La fonction envoie le caractère stocké dans `c` vers `stdout`. Bien que l'argument indiqué dans cette déclaration soit de type `int`, vous transmettez à la fonction une variable de type `char`. Vous pouvez aussi lui transmettre une variable de type `int`, du moment que sa valeur appartient à l'intervalle autorisé (0 à 255). La fonction renvoie le caractère qui vient d'être écrit ou `EOF` si elle a rencontré une erreur.

Le programme du Listing 14.2 que nous avons étudié contient une fonction `putchar()`. Celui du Listing 14.11 affiche les valeurs ASCII des caractères compris entre 14 et 127

Listing 14.10 : La fonction putchar()

```
/* La fonction putchar(). */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int count;

    for (count = 14; count < 128;)
        putchar(count++);

   exit(EXIT_SUCCESS);
}
```

Vous pouvez aussi afficher des chaînes de caractères avec la fonction `putchar()` (voir Listing 14.11).

Listing 14.11 : Affichage d'une chaîne de caractères avec putchar()

```
/* Utilisation de putchar() pour afficher des chaînes. */
#include <stdio.h>
#include <stdlib.h>

#define MAXSTRING 80

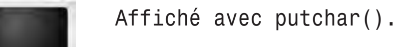
char message[] = "Affiché avec putchar().";
int main()
{
    int count;

    for (count = 0; count < MAXSTRING; count++)
    {

/* On cherche la fin de la chaîne. Quand on la trouve, on écrit    */
/* le caractère de retour à la ligne et on sort de la boucle. */

       if (message[count] == '\0')
       {
          putchar('\n');
          break;
       }
       else

/* Si ce n'est pas la fin de la chaîne, on écrit le prochain */
/* caractère. */
          putchar(message[count]);
    }
    exit(EXIT_SUCCESS);
}
```



```
Affiché avec putchar().
```

Les fonctions *putc()* et *fputc()*

Ces deux fonctions sont analogues, elles envoient un caractère vers le flot indiqué. `putc()` est l'implémentation macro de `fputc()` (les macros sont étudiées au Chapitre 21). La déclaration de `fputc()` a la forme suivante :

```
int fputc(int c, FILE *fp);
```

Le flot de sorties est transmis à la fonction par l'intermédiaire de l'argument `FILE *fp` (voir Chapitre 16). Si le flot indiqué est `stdout`, `fputc()` se comportera exactement comme `putchar()`. Les deux instructions suivantes sont donc équivalentes :

```
putchar('x');
fputc('x', stdout);
```

Utilisation de *puts()* et *fputs()* pour les sorties chaîne

Vos programmes auront plus souvent besoin d'afficher des chaînes que de simples caractères. la fonction de bibliothèque `puts()` permet d'afficher des chaînes de caractères. La fonction `fputs()` envoie la chaîne dans le flot indiqué. La déclaration de `puts()` a la forme suivante :

```
int puts(char *cp);
```

`*cp` est un pointeur vers le premier caractère de la chaîne que vous voulez afficher. La fonction `puts()` affiche la chaîne complète jusqu'au caractère qui précède le caractère nul de fin, et y ajoute le caractère de retour à la ligne. `puts()` renvoie une valeur positive si son exécution s'est déroulée correctement, ou `EOF` si une erreur s'est produite.

La fonction `puts()` peut afficher tout type de chaîne de caractères comme le montre le Listing 14.12.

Listing 14.12 : La fonction puts()

```
/* La fonction puts(). */
#include <stdio.h>
#include <stdlib.h>

/* Declare and initialize an array of pointers. */

char *messages[5] = { "Ceci", "est", "un", "message", "court." };

int main()
{
   int x;

   for (x=0; x<5; x++)
```

```
14:         puts(messages[x]);
15:
16:      puts("et cela est la fin !");
17:
18:      exit(EXIT_SUCCESS);
19:  }
```

```
Ceci
est
un
message
court.
et cela est la fin !
```

Analyse

Ce programme déclare un tableau de pointeurs. Ce type de tableau sera étudié au Chapitre 15. Les lignes 13 et 14 affichent chaque chaîne stockée dans le tableau `message`.

Utilisation de *printf()* et *fprintf()* pour les sorties formatées

Les fonctions de sorties précédentes n'affichent que des caractères ou des chaînes. Pour afficher des nombres, il faut utiliser les fonctions de sorties formatées de la bibliothèque C : `printf()` et `fprintf()`. Ces fonctions sont aussi capables d'afficher les caractères. Nous avons étudié la fonction `printf()` au Chapitre 7, et nous l'avons utilisée dans presque tous les exemples. Ce paragraphe vous fournira les ultimes détails.

Les deux fonctions `fprintf()` et `printf()` ont un fonctionnement analogue, mais `printf()` envoie toujours ses sorties dans `stdout`, alors que `fprintf()` les envoie vers un flot de sorties spécifié. En général, `fprintf()` est utilisée pour les sorties fichiers qui sont traitées au Chapitre 16.

La fonction `printf()` reçoit un nombre variable d'arguments. Le seul et unique argument obligatoire est la chaîne format, qui indique à `printf()` comment mettre en forme la sortie. Les arguments optionnels sont les variables et expressions que vous voulez afficher. Examinons ces quelques exemples simples qui vous donneront un aperçu des possibilités de `printf()` :

- L'instruction `printf("Hello, world !");` affiche le message "Hello, world !" à l'écran. Dans cet exemple, `printf()` a un unique argument, la chaîne format. Cette chaîne est une chaîne littérale qui sera affichée telle quelle à l'écran.
- L'instruction `printf("%d", i);` affiche la valeur de la variable entière `i` à l'écran. La chaîne format ne contient que la conversion %`d`, qui demande à `printf()` d'afficher un seul entier décimal. Le second argument, `i`, est le nom de la variable dont on veut afficher la valeur.

- L'instruction `printf("%d plus %d égal %d.", a, b, a+b);` affiche le message "2 + 3 égale 5" à l'écran (en supposant que a et b sont deux variables entières dont les valeurs sont respectivement 2 et 3). Dans cet exemple, `printf()` a quatre arguments : une chaîne format qui contient du texte littéral et des commandes de conversion, deux variables et une expression dont on veut afficher les valeurs.

La chaîne format de `printf()` peut être composée des éléments suivants :

- Une ou plusieurs commandes de conversion qui indiquent à `printf()` de quelle façon afficher une valeur appartenant à la liste d'arguments. Une commande de conversion est constituée du signe `%` suivi d'un ou plusieurs caractères.
- Des caractères qui ne font pas partie d'une commande de conversion et qui seront affichés tels quels.

Étudions en détail la commande de conversion. Les composants qui apparaissent entre crochets sont optionnels :

```
%[flag] [largeur_champ] [.[precision]] [l]carac_conversion
```

`carac conversion` est la seule partie obligatoire de cette commande (en dehors de `%`). Le Tableau 14.5 donne la liste de ces caractères de conversion avec leur signification.

Tableau 14.5 : Les caractères de conversion des fonctions printf() et fprintf()

Caractère de conversion	***Signification***
`d, i`	Affiche un entier signé en notation décimale.
`u`	Affiche un entier non signé en notation décimale.
`o`	Affiche un entier en notation octale non signée.
`x, X`	Affiche un entier en notation hexadécimale non signée. Utilisez `x` pour les sorties en minuscules et `X` pour les sorties majuscules.
`c`	Affiche un caractère (l'argument indique le code ASCII du caractère).
`e, E`	Affiche un nombre `float` ou `double` en notation scientifique (par exemple, 123,45 est affiché de cette façon : 1.234500e+002). L'affichage se fait avec six chiffres après la virgule, sauf si une autre précision est indiquée avec l'indicateur `f`. Utilisez `e`, ou `E` pour contrôler le type de caractères de la sortie.
`f`	Affiche un nombre `float` ou `double` en notation décimale (par exemple 123,45 sera affiché 123.450000). L'affichage se fait avec six chiffres aprèsaprès la virgule sauf si une autre précision est indiquée.
`g, G`	Utilise le format `e`, `E` ou `f`. Les formats `e` ou `E` sont choisis si l'exposant est inférieur à 3 ou supérieur à la précision (dont 6 est le défaut. Sinon le format `f` est utilisé). Les zéros sont tronqués.

Tableau 14.5 : Les caractères de conversion des fonctions printf() et fprintf() ***(suite)***

Caractère de conversion	***Signification***
`n`	Rien n'est affiché. L'argument qui correspond à une commande de conversion `n` est un pointeur de type `int`. La fonction `printf()` attribue à cette variable le nombre de caractères de la sortie.
`s`	Affiche une chaîne de caractères. L'argument est un pointeur de `char`. Les caractères sont affichés jusqu'au premier caractère nul rencontré ou jusqu'à ce que le nombre de caractères indiqué par précision soit atteint (par défaut ce nombre est 32767). Le caractère nul de fin n'est pas affiché.
`%`	Affiche le caractère %.

L'attribut `l` peut être placé devant le caractère de conversion. Cet attribut n'est disponible que pour les caractères de conversion `o`, `u`, `x`, `X`, `i`, `d`, `b`. Il signifie que l'argument est de type `long` plutôt que `int`. Quand cet attribut est associé aux caractères de conversion `e`, `E`, `f`, `g` et `G`, il signifie que l'argument est de type `double`.

L'indicateur de précision est constitué du point décimal (`.`) seul, ou accompagné d'un nombre. L'indicateur de précision ne s'applique qu'aux caractères de conversion `e E f g G s`. Il indique le nombre de chiffres à afficher après la virgule, ou le nombre de caractères, s'il est utilisé avec `s`. Le point décimal seul indique une précision de 0.

La largeur de champ indique le nombre minimum de caractères de la sortie. Cette largeur peut être représentée par :

- Un entier décimal ne commençant pas par 0. La sortie sera complétée à gauche par des blancs pour occuper la largeur de champ indiquée.
- Un entier décimal commençant par 0. La sortie sera alors complétée à gauche par des zéros pour occuper la largeur de champ indiquée.
- Le caractère (`*`). La valeur de l'argument suivant (qui sera de type `int`) sera interprétée comme la largeur de champ. Par exemple, si `w` est une variable de type `int` ayant une valeur de 10, l'instruction `printf("%*d", w, a);` affichera la valeur de `a` avec une largeur de champ de 10.

Si la largeur de champ n'est pas spécifiée, ou si elle est plus petite que la sortie, elle sera ajustée à la taille appropriée.

La dernière partie optionnelle de la chaîne format de `printf()` est le caractère de contrôle qui suit le caractère %. Ces caractères sont au nombre de quatre :

- – La sortie sera cadrée à gauche plutôt qu'à droite ; représente l'option par défaut.

- + Les nombres signés seront affichés avec leur signe (+ ou –).
- ’’ Un blanc signifie que les nombres positifs seront précédés d’un blanc.
- # Ce caractère ne s’applique qu’aux caractères de conversion `x`, `X` et `o`. Il indique que les nombres non nuls seront affichés avec `0X` ou `0x` devant (pour `x` et `X`), ou 0 pour les conversions de type `o`.

Vous pouvez coder la chaîne format de `printf()` de deux façons. La première consiste à la placer entre guillemets dans la liste d’arguments de `printf()`. La seconde est de la stocker en mémoire avec un caractère nul à la fin, et de transmettre son pointeur à la fonction. Par exemple :

```
char *fmt = "la réponse est %f.";
printf(fmt, x);
```

Ces instructions sont équivalentes à l’instruction suivante :

```
printf("La réponse est %f.", x);
```

Le Tableau 14.6 contient la liste des ordres de contrôle de la chaîne format les plus courants (voir Chapitre 7). L’ordre de contrôle `\n`, par exemple, permet d’afficher la sortie sur la ligne suivante.

Tableau 14.6 : Ordres de contrôle le plus souvent utilisés

Ordre	***Signification***
`\a`	Sonnerie
`\b`	Retour arrière
`\n`	Retour à la ligne
`\t`	Tabulation horizontale
`\\`	Antislash (backslash \)
`\?`	Point d’interrogation
`\'`	Guillemet simple
`\"`	Guillemet double

Avant de voir quelques exemples, nous vous mettons en garde contre une faille de sécurité potentielle dans vos programmes avec printf() et fprintf(). En effet, vous ne devez jamais indiquer en premier argument (le format) une chaîne de caractères sur laquelle l’utilisateur

peut avoir le contrôle. En d'autres termes, les deux lignes suivantes, qui affichent ce que l'utilisateur vient d'entrer au clavier, sont dangereuses :

```
fgets(buffer, sizeof(buffer), stdin);
printf(buffer);
```

Cet exemple fonctionne mais donne à l'utilisateur le contrôle du format et donc, s'il est malveillant, à bien plus... Les deux lignes précédentes doivent être écrites par exemple ainsi :

```
fgets(buffer, sizeof(buffer), stdin);
printf("%s", buffer);
```

Notez que maintenant, le format est contrôlé par le programme qui affichera les données sous forme de chaîne uniquement.

`printf()` est une commande sophistiquée. La meilleure façon d'apprendre à s'en servir, c'est d'étudier des exemples et de l'expérimenter. Le programme du Listing 14.13 illustre plusieurs manières d'utiliser cette fonction.

Listing 14.13 : La fonction printf()

```
/* Utilisation de printf(). */
#include <stdio.h>
#include <stdlib.h>

char *m1 = "Binaire";
char *m2 = "Decimal";
char *m3 = "Octal";
char *m4 = "Hexadecimal";

int main()
{

   float d1 = 10000.123;
   int n;


   puts("Affichage d'un nombre avec plusieurs largeurs \
         de champ.\n");
   printf("%5f\n", d1);
   printf("%10f\n", d1);
   printf("%15f\n", d1);
   printf("%20f\n", d1);
   printf("%25f\n", d1);
   puts("\n Appuyez sur Entrée pour continuer...");
   fflush(stdin);
   getchar();

   puts("\nOn utilise * pour obtenir la largeur de champ");
```

Listing 14.13 : La fonction printf() *(suite)*

```
    puts("d'une variable de la liste des arguments.\n");

    for (n=5; n <=25; n+=5)
       printf("%*f\n", n, d1);

    puts("\n Appuyez sur Entrée pour continuer...");
    fflush(stdin);
    getchar();

    puts("\nOn complète avec des zéros.\n");

    printf("%05f\n", d1);
    printf("%010f\n", d1);
    printf("%015f\n", d1);
    printf("%020f\n", d1);
    printf("%025f\n", d1);

    puts("\n Appuyez sur Entrée pour continuer...");
    fflush(stdin);
    getchar();

    puts("\nAffichage en octal, decimal, et hexadecimal.");
    puts("On utilise # à gauche des sorties octales et hex avec 0 et 0X.");
    puts("On utilise - à gauche pour justifier chaque valeur dans son champ.");
    puts("On affiche d'abord le nom des colonnes.\n");

    printf("%-15s%-15s%-15s", m2, m3, m4);

    for (n = 1; n < 20; n++)
       printf("\n%-15d%-#15o%-#15X", n, n, n, n);

    puts("\n Appuyez sur Entrée pour continuer...");
    fflush(stdin);
    getchar();

    puts("\n\nOn utilise la commande de conversion %n pour compter");
    puts("les caractères.\n");
    printf("%s%s%s%s%n", m1, m2, m3, m4, &n);

    printf("\n\nLe dernier printf() a affiché %d caractères.\n", n);

    exit(EXIT_SUCCESS);
 }
```

```
Affichage d'un nombre avec plusieurs largeurs de champ.

10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047

Appuyez sur Entrée pour continuer...
```

```
On utilise * pour obtenir la largeur de champ d'une variable de la
liste des arguments.

10000.123047
10000.123047
   10000.123047
        10000.123047
             10000.123047

Appuyez sur Entrée pour continuer...


On complète avec des zéros.

10000.123047
10000.123047
00010000.123047
0000000010000.123047
000000000000010000.123047

Appuyez sur Entrée pour continuer...


Affichage en octal, decimal, et hexadecimal.");
On utilise # à gauche des sorties octales et hex avec 0 et 0X.
On utilise - à gauche pour justifier chaque valeur dans son champ.
On affiche d'abord le nom des colonnes.

Décimal        Octal          Hexadécimal
 1             01             0X1
 2             02             0X2
 3             03             0X3
 4             04             0X4
 5             05             0X5
 6             06             0X6
 7             07             0X7
 8             010            0X8
 9             011            0X9
10             012            0XA
11             013            0XB
12             014            0XC
13             015            0XD
14             016            0XE
15             017            0XF
16             020            0X10
17             021            0X11
18             022            0X12
19             023            0X13

Appuyez sur Entrée pour continuer...

On utilise la commande de conversion %n pour compter les caractères

BinaireDecimalOctalHexadecimal

Le dernier printf() a affiché 30 caractères.
```

Redirection des entrées/sorties

Un programme qui travaille avec `stdin` et `stdout` peut utiliser une fonction du système d'exploitation appelée *redirection*. La redirection permet :

- D'envoyer les sorties de `stdout` dans un fichier disque plutôt que vers l'écran.
- De lire les entrées de `stdin` à partir d'un fichier disque plutôt qu'à partir du clavier.

La redirection n'est pas codée dans le programme. Vous devrez l'indiquer sur la ligne de commande quand vous exécuterez le programme. Avec Windows comme avec UNIX, les symboles de redirection sont : < et >. Étudions tout d'abord la redirection de la sortie.

Reprenons votre premier programme C, "hello.c". Ce programme envoie le message `Hello, world !` à l'écran avec la fonction `printf()`. Vous savez maintenant que `printf()` envoie ses sorties dans `stdout`, elles peuvent donc être redirigées. Quand vous entrez le nom du programme à l'invite de la ligne de commande, vous devez le faire suivre du symbole > puis du nom de la nouvelle destination :

```
hello > destination
```

Pour envoyer le message vers l'imprimante sur Windows, vous devez taper `hello > prn` (`prn` est le nom DOS de l'imprimante connectée sur le port LPT1:). Sur Unix, vous devez taper `hello | lpr` (il s'agit d'une redirection différente, lpr étant également un programme). Si vous tapez `hello > hello.txt`, le message sera sauvegardé dans le fichier `hello.txt`.

Soyez prudent lorsque vous redirigez une sortie vers un fichier disque. Si le fichier existe déjà, les données qu'il contient seront effacées pour être remplacées par votre sortie. Si le fichier n'existe pas, il sera créé. Le symbole >> pourra aussi être utilisé pour les redirections. Il indique que si le fichier destination existe déjà, la sortie doit être enregistrée à la suite des données qu'il contient. Le Listing 14.14 illustre la redirection.

Listing 14.14 : La redirection des entrées et des sorties

```
/* Exemple de redirection de stdin et stdout. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char buf[80];

   lire_clavier(buf, sizeof(buf));
   printf("L'entrée était : %s\n", buf);
   exit(EXIT_SUCCESS);
}
```

Ce programme reçoit une ligne du flot `sdtin` qu'il envoie dans `sdtout` en y ajoutant : `L'entrée était :`. Compilez ce programme puis exécutez-le sans utiliser la redirection (ce programme s'appelle List1414.c) en tapant `LIST1414` sur la ligne de commande. Si vous tapez `Le langage C en 21 jours` sur le clavier, le programme affichera :

```
L'entrée était : Le langage C en 21 jours
```

Si vous exécutez le programme en utilisant la commande `list1414 > test.txt` avec la même ligne de texte, rien ne sera affiché à l'écran. Le fichier `test.txt` a été créé sur disque, et il contient votre ligne de texte.

Vous constaterez que ce fichier ne contient que la ligne `L'entrée était : Le langage C en 21 jours`. Si vous aviez utilisé la commande `list1414 > prn` sur Windows ou `list1414 | lpr`, cette ligne aurait été imprimée sur l'imprimante.

Rediriger l'entrée

Examinons maintenant la redirection de l'entrée. Avec votre éditeur, créez le fichier source `input.txt` contenant une simple ligne de texte "`William Shakespeare`". Exécutez ensuite le Listing 14.14 en entrant la commande :

```
list1414 <input.txt
```

Le programme n'attendra pas que vous saisissiez du texte au clavier. Il va immédiatement afficher :

```
L'entrée était : William Shakespeare
```

Le flot `stdin` a été redirigé vers le fichier `input.txt`, la fonction `fgets()` de `lire clavier()` a donc lu une ligne de texte à partir du fichier plutôt que du clavier.

Vous pouvez rediriger les entrées et les sorties en même temps. Vous pouvez taper par exemple :

```
list1414 <input.txt >junk.txt
```

La lecture se fera à partir de `input.txt` et le résultat de l'exécution du programme sera sauvegardé dans `junk.txt`.

Souvenez-vous que la redirection de stdin et stdout est une fonction offerte par le système, elle est indépendante du langage C.

Quand utiliser *fprintf()*

La fonction de bibliothèque `fprintf()` est équivalente à `printf()`, à l'exception de ses sorties qu'elle envoie dans le flot indiqué. `fprintf()` est le plus souvent utilisée avec les fichiers disque qui sont étudiés au Chapitre 16. Il existe toutefois deux autres utilisations pour cette fonction.

Le flot *stderr*

`stderr` (pour *standard error*) est un des flots prédéfinis du langage C. Les messages d'erreur sont envoyés dans ce flot plutôt que dans `stdout`. Ce flot est, comme `stdout`, connecté à l'écran, mais de façon indépendante de celui-ci. En envoyant les messages d'erreur dans ce flot, on ne prend pas de risque si l'utilisateur a utilisé une redirection ; le message sera quand même affiché à l'écran :

```
fprintf(stderr, "une erreur s'est produite.");
```

Vous pouvez écrire une fonction pour traiter les messages d'erreur et l'appeler en cas d'erreur à la place de `fprintf()`.

```
message_erreur("une erreur s'est produite.");
void message_erreur(char *msg)
{
   fprintf(stderr, msg);
}
```

En utilisant votre propre fonction, vous apportez de la souplesse à votre code. Si vous avez besoin, dans certaines circonstances, d'envoyer les messages d'erreur dans un fichier plutôt qu'à l'écran, il suffira de modifier la fonction.



À faire

Créer des fonctions comme `message erreur` pour obtenir un code mieux structuré et plus facile à maintenir.

Utiliser `fprintf()` pour créer des programmes qui enverront leurs sorties vers `stdout`, `stderr` ou d'autres flots.

À ne pas faire

Utiliser `stderr` pour une fonction autre que les messages d'erreur ou d'avertissement.

Résumé

Vous avez appris, dans ce chapitre, que C utilise des flots pour les entrées/sorties des programmes. Les entrées/sorties sont traitées comme des séquences d'octets. Le langage C possède trois flots prédéfinis :

`stdin`	le clavier
`stdout`	l'écran

Les entrées clavier sont envoyées dans le flot `stdin`. En utilisant les fonctions de la bibliothèque du langage C, vous pouvez lire les entrées clavier caractère par caractère, ligne par ligne, ou comme des chaînes et des nombres formatés. Selon la fonction utilisée, les caractères entrés peuvent être stockés dans une mémoire tampon, ou être envoyés en écho sur l'écran.

Les sorties écran se font au travers du flot `stdout`. Comme les entrées, les sorties peuvent être traitées caractère par caractère, ligne par ligne, ou sous forme de chaînes et de nombres formatés.

Si votre programme utilise `stdin` et `stdout`, vous pouvez rediriger ses entrées et ses sorties. Les entrées peuvent provenir d'un fichier plutôt que du clavier, et les sorties peuvent être dirigées vers un fichier ou une imprimante plutôt que vers l'écran.

Enfin, vous avez découvert pourquoi les messages d'erreur devaient être envoyés dans le flot `stderr` plutôt que dans `stdout`. `stderr` étant connecté à l'écran de façon indépendante de `stdout`, l'utilisateur recevra ses messages d'erreur même s'il a redirigé les sorties du programme.

Q & R

Q Que se passera-t-il si j'envoie ma sortie vers une unité d'entrées ?

R Le programme ne marchera pas. Si vous essayez, par exemple, d'utiliser `stderr` avec la fonction `scanf()`, le programme sera compilé et vous obtiendrez le fichier exécutable. La sortie d'erreur étant incapable de fournir des entrées, le programme sera inopérant.

Q Que se passera-t-il si je redirige un des flots standards ?

R Cela pourra être la cause de futurs problèmes dans le programme. Si vous redirigez un flot, il faudra le réinitialiser s'il est utilisé de nouveau dans le programme. Beaucoup de fonctions décrites dans ce chapitre utilisent les flots standard. Si vous en modifiez-un, vous le modifiez pour tout le programme. Essayez, par exemple, d'attribuer stdin à stdout dans un des programmes du chapitre, et observez les résultats.

Q Y a-t-il un danger à utiliser des fonctions non ANSI dans un programme ?

R Beaucoup de bibliothèques de fonctions et certains compilateurs fournissent des fonctions très utiles qui ne font pas partie du standard ANSI/ISO ni d'un standard répandu comme POSIX ou BSD. Si vous ne prévoyez pas de changer de compilateur, et si vos programmes n'ont pas à tourner sur un autre matériel, il n'y a aucun problème. Vous serez concerné par la compatibilité ISO/ANSI ou POSIX si vous êtes amené à utiliser d'autres compilateurs ou un autre type de matériel.

Q Pourquoi ne pas utiliser `fprintf()` à la place de `printf()` ?

R Si vous travaillez avec les flots d'entrées/sorties standard, utilisez `printf()` et `scanf()`. En utilisant ces fonctions simples, vous n'avez pas besoin de vous préoccuper des autres flots.

Atelier

Cet atelier présente un quiz destiné à consolider les connaissances acquises dans ce chapitre et quelques exercices pour mettre en pratique ce que vous venez d'apprendre.

Quiz

1. Qu'est-ce qu'un flot ?
2. Les unités suivantes sont-elles destinées aux entrées ou aux sorties ?
   a. Clavier.
   b) Écran.
   c) Disque.
3. Donnez la liste des trois flots prédéfinis et des unités qui leur sont associées.
4. Quels sont les flots utilisés par les fonctions suivantes ?
   a) `printf()`.
   b) `puts()`.
   c) `scanf()`.
   d) `fprintf()`.
5. Quelle est la différence entre les entrées caractère de `stdin` qui utilisent la mémoire tampon, et celles qui ne l'utilisent pas ?
6. Quelle est la différence entre les entrées caractère de `stdin` qui sont recopiées, et celles qui ne le sont pas ?

7. Pouvez-vous récupérer plus d'un caractère à la fois avec la fonction `ungetc()` ? Pouvez-vous récupérer le caractère `EOF` ?
8. Comment est déterminée la fin de ligne quand vous utilisez les fonctions d'entrées ligne du C ?
9. Dans la liste suivante, quelles sont les conversions correctes ?
   a) `"%d"`.
   b) `"%4d"`.
   c) `"%3i%c"`.
   d) `"%q%d"`.
   e) `"%%%i"`.
   f) `"%9ld"`.
10. Quelle est la différence entre `stderr` et `stdout` ?

Exercices

1. Écrivez l'instruction qui affichera le message `"hello, world"` à l'écran.
2. Refaites l'exercice 1 avec deux autres fonctions.
3. Écrivez l'instruction qui lira une chaîne de 30 caractères au maximum et qui tronquera la ligne si elle rencontre un astérisque.
4. Écrivez l'instruction qui affichera :

```
Jack demande, "qu'est-ce qu'un antislash ?"
Jill répond, "c'est '\'"
```

5. **TRAVAIL PERSONNEL :** Écrivez un programme utilisant la redirection pour lire un fichier, compter le nombre d'occurrences dans ce fichier pour chaque lettre, puis afficher les résultats à l'écran.
6. **TRAVAIL PERSONNEL :** Écrivez un programme qui lira les entrées clavier et les reproduira sur l'écran. Ce programme devra compter les lignes. Créez une clé de fonction pour sortir du programme.

Tour d'horizon de la Partie III

Les deux premières parties de ce livre vous ont fourni les bases du langage C. Au cours de cette troisième partie, nous allons apprendre à tirer meilleur parti du langage C. Nous rassemblerons les connaissances acquises et verrons comment les mettre en pratique.

Lorsque vous aurez terminé les sept chapitres qui forment cette troisième partie, vous pourrez voler de vos propres ailes.

Qu'allez-vous voir maintenant ?

Voici les grands thèmes de cette troisième partie :

Le Chapitre 15 : *Retour sur les pointeurs* va vous permettre d'approfondir un des points les plus délicats du C.

Le Chapitre 16 : *Utilisation de fichiers sur disque* traite d'un sujet qui prend toute son importance dès qu'on aborde les applications réelles.

Les Chapitres 17, 18 et 19 : *Manipulations de chaînes de caractères*, *Retour sur les fonctions* et *Exploration de la bibliothèque des fonctions* vous bombarderont d'une multitude de fonctions : de celles qui donnent au C toute sa puissance et toute sa souplesse.

Au Chapitre 20, *La mémoire*, nous allons étudier en profondeur la gestion mémoire.

Enfin, le Chapitre 21, intitulé *Comment tirer parti du préprocesseur* sera la cerise sur le gâteau car, en plus de cette partie importante du langage, nous allons découvrir comment passer des arguments, depuis la ligne de commande jusqu'au programme.

15

Retour sur les pointeurs

Au Chapitre 9, nous vous avons présenté les notions de base concernant les pointeurs. Si nous avons insisté sur ce sujet, c'est parce que c'est l'un des domaines les plus importants du langage C. Nous allons y revenir pour étudier en particulier :

- Comment déclarer un pointeur vers un pointeur
- Comment utiliser les pointeurs avec des tableaux à plusieurs dimensions
- Comment déclarer des tableaux de pointeurs
- Comment déclarer des pointeurs vers des fonctions
- Comment utiliser les pointeurs pour créer des listes liées pour l'enregistrement des données

Pointeur vers un pointeur

Nous savons, depuis le Chapitre 9, qu'un pointeur est une variable numérique dont la valeur représente l'adresse d'une autre variable. Un pointeur se déclare en utilisant l'opérateur d'indirection (*). Ainsi :

```
int *ptr;
```

déclare un pointeur appelé ptr pouvant pointer vers une variable de type int. Vous pouvez utiliser l'opérateur d'adresse (&) pour placer dans ptr l'adresse d'une variable particulière du type convenable (ici int) :

```
ptr = &x;
```

Par cette instruction, vous rangez dans ptr l'adresse de la variable x. On dit alors que ptr pointe vers x. Au moyen de l'opérateur d'indirection *, vous pouvez alors manipuler le contenu de cette variable x. Les deux instructions suivantes donnent la valeur 12 à x :

```
x = 12;
*ptr = 12;
```

Comme un pointeur est lui-même une variable numérique, il est situé dans la mémoire de l'ordinateur à une adresse particulière. Dès lors, rien n'empêche de créer un pointeur vers un pointeur, c'est-à-dire une variable dont la valeur est l'adresse d'un pointeur. Voici comment :

```
int x = 12;              /* x est une variable de type int */
int *ptr = &x;           /* ptr est un pointeur vers x */
int **ptr_to_ptr = &pre; /* ptr_to_ptr est un pointeur vers
                            un pointeur de type int */
```

Notez l'utilisation d'un double opérateur d'indirection "**" lorsqu'on déclare un pointeur vers un pointeur. De même, vous utiliserez cette double indirection quand vous voudrez vous référer au contenu de la variable de type int. Ainsi, l'instruction

```
**ptr_to_ptr = 12;
```

donne, elle aussi, la valeur 12 à la variable x et l'instruction

```
printf("%d", **ptr_to_ptr);
```

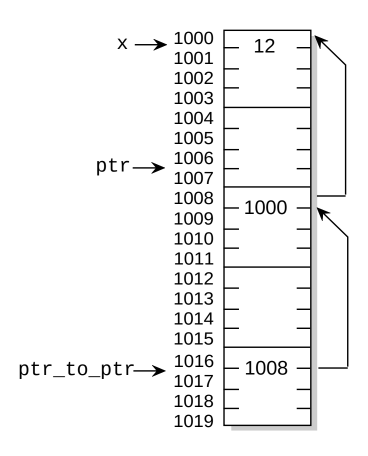
affichera bien la valeur de x. Oublier l'un des opérateurs d'indirection vous conduirait à une erreur, comme dans l'instruction :

```
*ptr_to_ptr = 12;
```

qui assigne la valeur 12 à `ptr`. La variable numérique 12 n'a aucun sens comme pointeur et le résultat sera erroné.

Lorsque vous déclarez un pointeur vers un pointeur, on dit qu'il y a *indirection multiple*. Les relations entre une variable, un pointeur et un pointeur vers un pointeur sont illustrées par la Figure 15.1. Il n'y a réellement aucune limite (si ce n'est celle du simple bon sens) à la *profondeur* de cette indirection. En pratique, on dépasse rarement le niveau 2.

Figure 15.1
Illustration de la notion de "pointeur vers un pointeur"



À quoi peuvent servir les pointeurs vers des pointeurs ? L'application la plus fréquente est l'utilisation de tableaux de pointeurs que nous étudierons plus loin, dans ce même chapitre. Le Listing 19.5 vous en montrera, au Chapitre 19, une application.

Pointeurs et tableaux à plusieurs dimensions

Au Chapitre 8, nous avons parlé des relations existant entre les pointeurs et les tableaux. Le nom d'un tableau non suivi de crochets représente un pointeur vers le premier élément de ce tableau. Il est bien plus facile d'utiliser la notation avec pointeur lorsqu'on veut accéder à certains types de tableaux. Au Chapitre 8, nous nous étions limités à des tableaux à une seule dimension. Que se passe-t-il lorsqu'ils ont deux dimensions et plus ?

Souvenez-vous qu'un tableau à plusieurs dimensions se déclare en indiquant la valeur de chacune de ses dimensions. Ainsi, la déclaration suivante crée un tableau de huit variables de type `int`, réparties en deux groupes de quatre (deux lignes de quatre colonnes, si vous préférez) :

```
int multi[2][4];
```

La Figure 15.2 illustre ce découpage :

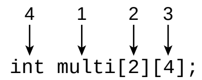


Figure 15.2
Éléments d'une décla-ration de tableau à plusieurs dimensions.

Ce qui s'interprète de la façon suivante :

1. On déclare un tableau appelé `multi`.
2. Ce tableau contient deux éléments principaux.
3. Chacun de ces deux éléments contient à son tour quatre éléments.
4. Chacun de ces éléments est de type `int`.

Une déclaration de tableau à plusieurs dimensions se lit de gauche à droite (ce qui ne devrait guère bouleverser vos habitudes).

En utilisant cette notion de groupe contenant des groupes (ou de tableau contenant des tableaux), on aboutit à la représentation imagée de la Figure 15.3.

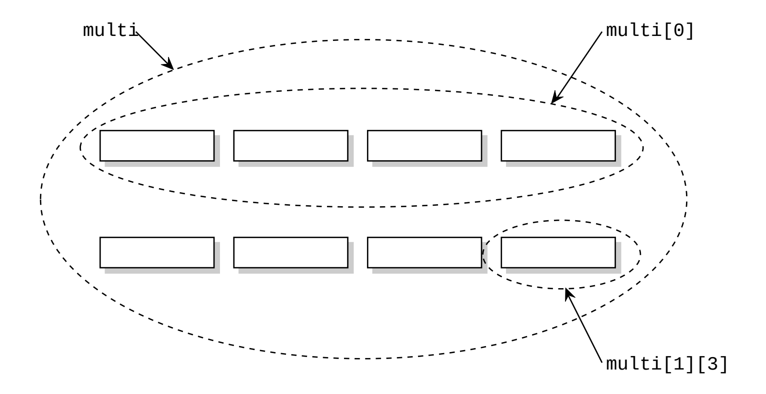


Figure 15.3
Un tableau à deux dimensions vu comme un tableau de tableaux.

Revenons maintenant à la notion de noms de tableaux considérés comme des pointeurs. Comme pour les tableaux à une seule dimension, le nom d'un tableau à plusieurs dimensions est un pointeur vers le premier élément du tableau. Toujours avec le même exemple du tableau `multi`, on peut dire que `multi` est un pointeur vers le premier élément d'un tableau à deux dimensions déclaré comme `int  multi[2][4]`. Quel est exactement le premier élément de `multi` ? Comme il s'agit d'un tableau de tableaux, ce n'est sûrement pas `multi[0][0]`. C'est, en réalité, `multi[0]`, c'est-à-dire le premier "sous-tableau" de quatre variables de type `int`.

Mais, `multi[0]` est-il un tableau ou un pointeur ? Si c'est un pointeur, il doit bien pointer vers quelque chose. En effet, il pointe vers le premier élément de ce sous-tableau : `multi[0][0]`. Pourquoi `multi[0]` est-il un pointeur ? Souvenez-vous qu'un nom de tableau non suivi de crochets représente un pointeur vers le premier élément de ce tableau. Ici, il manque le second groupe de crochets ; on peut donc considérer que `multi[0]` est un pointeur.

Si vos idées ne sont pas très claires à ce sujet, ne vous inquiétez pas outre mesure. C'est un concept quelque peu inhabituel qui, au début, est difficile à appréhender. Les règles suivantes à propos des tableaux à *n* dimensions devraient vous aider à y voir plus clair :

- Le nom d'un tableau suivi de *n* paires de crochets (chacune contenant naturellement un indice approprié) est un tableau de données.
- Le nom d'un tableau suivi de moins de n paires de crochets est un pointeur vers un sous-tableau.

Dans notre exemple, `multi` est donc un pointeur, tout comme `multi[0]`, alors que `multi[0][0]` est une variable numérique de type `int`.

Voyons maintenant vers quoi pointent ces pointeurs. Le programme du Listing 15.1 déclare un tableau à deux dimensions (semblable à celui que nous venons d'utiliser), et imprime les pointeurs et les adresses des premiers éléments.

Listing 15.1 : Relations entre pointeurs et tableaux à plusieurs dimensions

```
/* Pointeurs et tableaux à plusieurs dimensions. */
#include <stdio.h>
#include <stdlib.h>

int multi[2][4];

int main()
{
   printf("\nmulti = %p", multi);
   printf("\nmulti[0] = %p", multi[0]);
   printf("\n&multi[0][0] = %p\n", &multi[0][0]);
   exit(EXIT_SUCCESS);
  }
```

```
multi = 0x8049620
multi[0] = 0x8049620
&multi[0][0] = 0x8049620
```

Analyse

On constate que les valeurs sont bien identiques.

Intéressons-nous maintenant à la *taille* de ces éléments. Le Listing 15.2 va nous permettre de la comparer.

Listing 15.2 : Détermination de la taille des éléments

```
/* Taille d'éléments de tableaux à plusieurs dimensions. */
#include <stdio.h>
#include <stdlib.h>

int multi[2][4];

int main()
{
  printf("\nLa taille de multi est égale à %u", sizeof(multi));
  printf("\nLa taille de multi[0] est égale à %u",
         sizeof(multi[0]));
  printf("\nLa taille de multi[0][0] est égale à %u\n",
         sizeof(multi[0][0]));
  exit(EXIT_SUCCESS);
}
```

On obtient, sur un système 32 bits, les résultats suivants :

```
La taille de multi est égale à 32
La taille de multi[0] est égale à 16
La taille de multi[0][0] est égale à 4
```

Analyse

Ce ne sont pas les valeurs elles-mêmes qui comptent, mais leur *rapport*. On voit clairement que `multi[0][0]`, qui est une simple variable numérique, a une certaine taille (4octets). Le premier sous-tableau, `multi[0]`, représente un groupement de quatre variables, et a donc une taille quatre fois supérieure. Le tableau `multi` a une taille lui permettant de contenir deux de ces sous-tableaux, soit 2 x 4 = 8 fois la taille d'une variable numérique élémentaire.

Le compilateur C "connaît" la taille de l'objet pointé et en tient compte dans les calculs d'adresses. Lorsque vous incrémentez un pointeur, sa valeur est augmentée proportionnellement à la taille de l'objet sur lequel il pointe. Ainsi, `multi` étant un sous-tableau de 4 éléments de quatre octets chacun (longueur d'un `int`), si on incrémente de pointeur d'une unité, sa valeur numérique augmentera de 16. Si `multi` pointe sur `multi[0]`, (`multi + 1`) doit pointer sur `multi[1]`. C'est ce que montre le programme du Listing 15.3.

Listing 15.3 : Arithmétique des pointeurs et tableaux à plusieurs dimensions

```
/* Arithmétique des pointeurs et tableaux à plusieurs dimensions. */
 #include <stdio.h>
 #include <stdlib.h>

```

```
int multi[2][4];

int main()
{
  printf("\nLa valeur de (multi) est %p", multi);
  printf("\nLa valeur de (multi + 1) est %p", (multi+1));
  printf("\nL'adresse de multi[1] est %p\n", &multi[1]);
  exit(EXIT_SUCCESS);
}
```



```
La valeur de (multi) est 0x8049660
La valeur de (multi + 1) est 0x8049660
L'adresse de multi[1] est 0x8049660
```

Analyse

Si vous incrémentez `multi` de 1, sa valeur augmente en réalité de quatre fois la taille d'une variable de type `int`.

Dans cet exemple, `multi` est un pointeur vers `multi[0]`, et `multi[0]` un pointeur vers `multi[0][0]`. Donc, `multi` est un pointeur vers un pointeur. Pour imprimer la *valeur* qui se trouve en `multi[0][0]`, vous avez le choix entre les trois instructions suivantes :

```
printf("%d", multi[0][0]);
printf("%d", *multi[0]);
printf("%d", **multi);
```

Cela s'applique aux tableaux ayant plus de deux dimensions. Un tableau à trois dimensions est un tableau dont les éléments sont des tableaux à deux dimensions. Chacun de ces derniers est, à son tour, composé de tableaux à une dimension.

Jusqu'ici, nous avons utilisé des indices exprimés par des constantes, mais rien n'empêche d'utiliser des variables. Reprenons notre tableau `multi` :

```
int multi[2][4];
```

Pour déclarer un pointeur `ptr` pointant vers un élément de `multi` (c'est-à-dire pointant vers un sous-tableau de 4 éléments de type `int`), on peut écrire :

```
int (*ptr)[4];
```

Pour qu'il pointe vers le premier sous-tableau, on écrit :

```
ptr = multi;
```

Peut-être vous interrogez-vous sur la raison d'être des parenthèses dans la déclaration de `ptr` ? C'est une question de priorité. Les crochets `[]` ont une plus forte priorité que l'opérateur d'indirection `*`. Si nous écrivions :

```
int *ptr[4];
```

nous définirions un tableau de quatre pointeurs vers des variables de type `int` ; ce n'est pas ce que nous voulons.

Comment utiliser des pointeurs vers des éléments de tableaux à plusieurs dimensions ? Comme s'il s'agissait de tableaux à une seule dimension ; par exemple, pour passer l'adresse d'un tableau à une fonction, ainsi que le montre le Listing 15.4.

Listing 15.4 : Comment passer un tableau à plusieurs dimensions à une fonction au moyen d'un pointeur

```
/* Passer un tableau à plusieurs dimensions à une fonction
   au moyen d'un pointeur */
#include <stdio.h>
#include <stdlib.h>

void printarray_1(int (*ptr)[4]);
void printarray_2(int (*ptr)[4], int n);

int main()
{
    int  multi[3][4] = { { 1,  2,  3,  4 },
                         { 5,  6,  7,  8 },
                         { 9, 10, 11, 12 } };
    /* ptr est un pointeur vers un tableau de 4 int. */

    int (*ptr)[4], count;

    /* Maintenant, ptr va pointer vers le premier
       élément de multi. */

    ptr = multi;

    /* A chaque tour de la boucle, ptr est incrémenté pour pointer
       sur l'élément suivant (le sous-tableau de 4 éléments). */

    for (count = 0; count < 3; count++)
       printarray_1(ptr++);

    puts("\n\nAppuyez sur Entrée...");
    getchar();
    printarray_2(multi, 3);
    printf("\n");
    exit(EXIT_SUCCESS);
}

void printarray_1(int (*ptr)[4])
{
    /* Affiche les éléments d'un tableau de 4 entiers.
       p est un pointeur de type int. Il est nécessaire
```

```
40:         de caster p pour qu'il soit égal à l'adresse
41:         contenue dans ptr. */
42:
43:      int *p, count;
44:      p = (int *)ptr;
45:
46:      for (count = 0; count < 4; count++)
47:       printf("\n%d", *p++);
48:  }
49:
50:  void printarray_2(int (*ptr)[4], int n)
51:  {
52:      /* Affiche les éléments d'un tableau d'entiers
53:         de n groupes de 4 éléments. */
54:
55:      int *p, count;
56:      p = (int *)ptr;
57:
58:      for (count = 0; count < (4 * n); count++)
59:         printf("\n%d", *p++);
60:  }
```

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
Appuyez sur Entrée...

 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
```

Analyse

Le programme déclare un tableau d'entiers, `multi[3][4]` aux lignes 11 à 13. En outre, il contient deux fonctions, `printarray1()` et `printarray2()`, qui vont nous servir à afficher le tableau.

La première de ces fonctions (lignes 36 à 48) ne reçoit qu'un seul argument, qui est un pointeur vers un tableau de quatre entiers. Elle affiche les quatre éléments de ce tableau. La première fois, `main()` appelle `printarray1()` à la ligne 27 en lui passant un pointeur vers le premier élément (le premier sous-tableau de 4 entiers) de `multi`. Elle appelle ensuite `printarray1()` deux autres fois en auto-incrémentant simplement `ptr` qui, dès lors, va pointer successivement sur le deuxième sous-tableau puis sur le troisième. À la suite de ces trois appels, la totalité de `multi` aura été affichée.

La seconde fonction `printarray2()` utilise une approche différente. Elle reçoit, elle aussi, un pointeur vers un tableau de quatre entiers mais, en outre, un entier lui indique le nombre de ces tableaux contenus dans `multi`. Avec un seul appel (ligne 32), `printarray2()` va afficher la totalité de `multi`.

Les deux fonctions mettent en pratique la notation des pointeurs pour avancer dans les éléments du tableau. La notation `(* int) ptr` utilisée dans les deux fonctions (lignes 43 et 55) ne vous semble peut-être pas assez claire. Il s'agit d'un casting (d'une *coercition*, comme on dit parfois en français) qui change temporairement le type des données de la variable, les faisant passer du type déclaré à un nouveau type. Il est nécessaire ici, parce que `ptr` et `p` sont des pointeurs de type différent (`p` est un pointeur vers un `int`, alors que `ptr` est un pointeur vers un tableau de quatre `int`). Tout se passe comme si on disait au compilateur "Pour cette instruction, tu vas considérer que `ptr` est un pointeur vers un `int`". Nous reviendrons sur la coercition au Chapitre 20.



À ne pas faire

Oublier d'utiliser un double opérateur d'indirection (`**`) lorsqu'on déclare un pointeur vers un pointeur.

Oublier qu'un pointeur s'incrémente en fonction de la taille de l'objet sur lequel il pointe.

Oublier d'utiliser des parenthèses lorsqu'on déclare un processus vers un tableau. Pour déclarer un pointeur vers un tableau de caractères, utilisez la forme suivante :

```
char (*lettres)[26];
```

Pour déclarer un tableau de pointeurs vers des caractères, utilisez :

```
char *lettres[26];
```

Tableaux de pointeurs

Nous avons vu au Chapitre 8 qu'un tableau est une collection d'adresses de rangement de même type, auxquelles on se réfère sous le même nom. Comme, en C, les pointeurs constituent un type de données, il n'y a aucune raison pour qu'on ne puisse pas constituer des tableaux de pointeurs.

L'usage le plus fréquent de ce type de construction concerne les chaînes de caractères. Comme vous l'avez appris au Chapitre 10, une chaîne de caractères est une suite de caractères rangés en mémoire. Le début de la chaîne est indiqué par un pointeur vers le premier caractère (un pointeur de type `char`, naturellement) et la fin de la chaîne est marquée par le caractère `NULL`. En déclarant et en initialisant un tableau de pointeurs vers des types `char`, vous pouvez accéder à un grand nombre de chaînes et les manipuler. Chaque élément du tableau pointe en effet sur une chaîne différente, il suffit donc de boucler sur ce tableau pour y accéder tour à tour.

Chaînes et pointeurs – révision

C'est le moment de revoir quelques-unes des notions abordées au Chapitre 10 concernant l'allocation des chaînes et leur initialisation. Pour allouer et initialiser une chaîne, vous devez déclarer un tableau de type `char` de la façon suivante :

```
char message[] = "Ceci est un message";
```

ou bien, en utilisant un pointeur :

```
char *message = "Ceci est un message";
```

Les deux déclarations sont équivalentes. Dans les deux cas, le compilateur alloue assez de place pour contenir la chaîne et son terminateur ; `message` est un pointeur vers le début de la chaîne. Que pensez-vous, maintenant, des deux instructions suivantes ?

```
char message1[20];
char *message2;
```

La première instruction déclare un tableau de type `char` ayant une longueur de vingt caractères ; `message1` est alors un pointeur vers le début de ce tableau. La place a été allouée, mais rien n'y a été rangé : la chaîne n'est pas initialisée.

La seconde instruction déclare un pointeur de type `char`, `message2`, et c'est tout. Aucune place n'a été réservée et, a fortiori, rien n'a été initialisé. Pour créer une chaîne de caractères sur laquelle `message2` pointerait, vous devez commencer par allouer de la mémoire pour la chaîne. Au Chapitre 10, vous avez appris à utiliser pour cela la fonction `malloc()`.

Souvenez-vous qu'il faut allouer de la place pour toute chaîne, soit à la compilation, soit à l'exécution, avec une fonction de type `malloc()`.

Tableau de pointeurs de type char

Maintenant que vous avez bien en tête ces notions, comment allez-vous déclarer un tableau de pointeurs ? L'instruction suivante déclare un tableau de dix pointeurs de type `char` :

```
char *message[10];
```

Chaque élément du tableau `message[]` est un pointeur individuel de type `char`. Comme vous l'avez sans doute deviné, vous pouvez associer la déclaration et l'initialisation :

```
char message[10] = {"un", "deux", "trois"};
```

Voici les effets de cette déclaration :

- Elle alloue un tableau de dix éléments appelé `message`, chaque élément du `message` étant un pointeur de type `char`.
- Elle alloue de la place quelque part en mémoire (peu importe où) et y place des chaînes d'initialisation avec, pour chacune, un terminateur `NULL`.
- Elle initialise `message[0]` pour qu'il pointe sur le premier caractère de la chaîne 1, `message[1]` pour qu'il pointe sur le premier caractère de la chaîne 2 et `message[2]`, pour qu'il pointe sur le premier caractère de la chaîne 3.

C'est ce qu'illustre la Figure 15.4, sur laquelle on voit les relations existant entre le tableau de pointeurs et les chaînes de caractères. Notez que, dans cet exemple, les éléments `message[3]` à `message[9]` ne sont pas initialisés et ne pointent donc sur rien du tout.

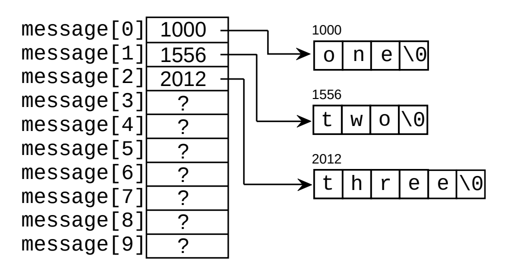


Figure 15.4
Un tableau de pointeurs de type char.

Portons maintenant notre attention sur le Listing 15.5 qui montre un exemple d'utilisation d'un tableau de pointeurs.

Listing 15.5 : Initialisation et utilisation d’un tableau de pointeurs de type char

```
 1:   /* Initialisation d'un tableau de pointeurs de type char. */
 2:   #include <stdio.h>
 3:   #include <stdlib.h>
 4:
 5:   int main()
 6:   {
 7:      char *message[8] = {"Lorsque", "l'enfant", "paraît,", "le",
 8:                          "cercle", "de", "famille", "s'agrandit."};
 9:      int count;
10:
11:     printf("\n");
12:     for (count = 0; count < 8; count++)
13:        printf("%s ", message[count]);
14:     printf("\n");
15:     exit(EXIT_SUCCESS);
16:   }
```



```
Lorsque l'enfant paraît, le cercle de famille s'agrandit.
```

Analyse

Dans le programme, on déclare un tableau de huit pointeurs de type `char` initialisés pour qu’ils pointent sur 8 chaînes reproduisant, à peu de chose près, un vers de Victor Hugo (lignes 7 et 8). Une boucle `for` (lignes 12 et 13) affiche les éléments du tableau en intercalant un espace entre chacun d’eux.

Vous voyez que la manipulation des tableaux de pointeurs est plus facile que celle des chaînes elles-mêmes. Cet avantage est évident dans des programmes plus compliqués comme celui qui va vous être présenté dans ce chapitre, et dans lequel on appelle une fonction. Il est, en effet, bien plus facile de passer un pointeur à une fonction que de lui passer plusieurs chaînes. Ce programme (Listing 15.6) est une réécriture du programme précédent dans lequel l’affichage se fait en appelant une fonction.

Listing 15.6 : Passer un tableau de pointeurs à une fonction

```
/* Passer un tableau de pointeurs à une fonction. */
#include <stdio.h>
#include <stdlib.h>

void print_strings(char *p[], int n);

int main()
{
     char *message[8] = {"Lorsque", "l'enfant", "paraît,", "le",
                         "cercle", "de", "famille", "s'agrandit."};

```

Listing 15.6 : Passer un tableau de pointeurs à une fonction *(suite)*

```
12:       print_strings(message, 8);
13:       exit(EXIT_SUCCESS);
14:   }
15:   void print_strings(char *p[], int n)
16:   { int count;
17:
18:     for (count = 0; count < n; count++)
19:         printf("%s ", p[count]);
20:     printf("\n");
21:   }
```



```
Lorsque l'enfant paraît, le cercle de famille s'agrandit.
```

Analyse

Comme on peut le voir à la ligne 15, la fonction `print strings()` demande deux arguments : le premier est un tableau de pointeurs de type `char` et le second, le nombre d'éléments à afficher. Rien de plus à signaler.

Vers le début de cette section, nous vous avions annoncé une démonstration ultérieure. Eh bien, vous venez de l'avoir (et de la voir !).

Un exemple

Il est grand temps, maintenant, de passer à quelque chose de plus compliqué. Le programme du Listing 15.7 met en œuvre plusieurs des notions déjà étudiées dont, en particulier, les tableaux de pointeurs. L'utilisateur tape des lignes de texte au clavier, le programme les range en mémoire et en garde une trace grâce à un tableau de pointeurs. Lorsque l'utilisateur tape une ligne vierge, le programme trie les lignes entrées et les affiche à l'écran.

Voici l'architecture de ce programme, vu sous l'angle de la programmation structurée :

1. Lire des lignes au clavier, une par une, jusqu'à rencontrer une ligne vierge.
2. Trier les lignes en ordre alphabétique ascendant.
3. Afficher les lignes triées.

Cette liste suggère l'écriture de trois fonctions distinctes, une pour chaque tâche. Elles s'appelleront respectivement `get lines()`, `sort()` et `print strings()`.

La première, `get lines()`, peut se décomposer ainsi :

1. Garder une trace du nombre de lignes entrées par l'utilisateur et renvoyer cette valeur au programme appelant, une fois toutes les lignes entrées.

2. Limiter le nombre de lignes pouvant être tapées par l'utilisateur, en fixant par avance un maximum.
3. Allouer de la place en mémoire pour chaque ligne.
4. Constituer un tableau de pointeurs contenant les adresses des lignes entrées par l'utilisateur.
5. Revenir au programme appelant si l'utilisateur tape une ligne vierge.

La deuxième, `sort()`, est une fonction de tri très simplifiée avec laquelle on balaye le tableau des lignes entrées, en partant de la première et en effectuant des permutations. À la fin du premier balayage, la plus petite (par le code des caractères et non par le nombre de caractères tapés) se retrouve en tête. On part ensuite de la deuxième ligne et on trie le tableau en la comparant aux *n* – *2* lignes restantes. À la fin de ce balayage, les deux plus petites lignes sont en place. On continue ainsi jusqu'à ce qu'il ne reste plus qu'une seule ligne, qui est forcément la plus grande.

C'est sans doute méthode de tri la plus lente mais, en raison de la simplicité de son algorithme, elle est presque aussi célèbre que "Hello, world" ou "Bonjour le monde". Notez que, en réalité, ce ne sont pas les lignes qu'on permute, mais les pointeurs sur les lignes.

Vous verrez, au Chapitre 19, que la bibliothèque standard C contient une fonction bien plus élaborée et bien plus rapide, `qsort()`. Mais, pour un exemple aussi court que celui-ci, notre méthode est suffisante.

La dernière fonction, `print strings()`, existe déjà : nous venons de la rencontrer dans le Listing 15.6.

Listing 15.7 : Tri et affichage d'un groupe de lignes entrées au clavier

```
/* L'utilisateur tape une suite de phrases au clavier, elles
   sont triées puis affichées à l'écran. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINES 25     /* pas plus de 25 phrases */

int get_lines(char *lines[]);         /* entrée des phrases */
void sort(char *p[], int n);          /* tri des phrases */
void print_strings(char *p[], int n); /* réaffichage  à l'écran */

char *lines[MAXLINES];

int main()
{
    int number_of_lines;
```

Listing 15.7 : Tri et affichage d’un groupe de lignes entrées au clavier *(suite)*

```

     /* Lire les phrases au clavier. */

     number_of_lines = get_lines(lines);

     if (number_of_lines < 0)
     { puts("Erreur d'allocation mémoire");
       exit(EXIT_FAILURE);
     }

     sort(lines, number_of_lines);
     print_strings(lines, number_of_lines);
     return(0);
}

int get_lines(char *lines[])
{ int n = 0;
  char buffer[80];  /* Mémoire de stockage temporaire */

  puts("Tapez les phrases une par une.");
  puts("Terminez par un simple appui sur Entrée.");

  while ((n < MAXLINES) &&
         (lire_clavier(buffer, sizeof(buffer)) != 0))
  { if ((lines[n] = malloc(strlen(buffer)+1)) == NULL)
       return -1;
    strcpy(lines[n++], buffer);
  }
  return n;

} /* Fin de get_lines() */

void sort(char *p[], int n)
{ int a, b;
  char *x;

  for (a = 1; a < n; a++)
   for (b = 0; b < n-1; b++)
   { if (strcmp(p[b], p[b+1]) > 0)
     { x = p[b];
       p[b] = p[b+1];
       p[b+1] = x;
     }
   }
}  /* Fin de sort() */

void print_strings(char *p[], int n)
{ int count;

  for (count = 0; count < n; count++)
     printf("%s\n ", p[count]);
}  /* Fin de print_strings() */
```

Voici un exemple de ce que l'on peut obtenir :

```
Tapez les phrases une par une.
Terminez par un simple appui sur Entrée.
chien
ordinateur
assiette
jeu
fourchette
zoo

assiette
chien
fourchette
jeu
ordinateur
zoo
```

Analyse

Nous allons examiner quelques détails de ce programme dans lequel on voit apparaître de nouvelles fonctions, et, en particulier, le fichier d'en-tête string.h qui contient les prototypes des fonctions de manipulation de chaînes de caractères.

Dans la fonction `get lines()`, l'entrée des lignes est contrôlée par les instructions des lignes 41 et 42 qui testent deux conditions : on n'a pas dépassé le nombre de lignes maximum (`MAXLINES`), et `lire clavier()` n'a pas renvoyé `0` (chaîne vide).

Lorsqu'une ligne a été lue au clavier, il faut allouer de la place pour la ranger. C'est ce que fait l'instruction de la ligne 43 :

```
if ((lines[n] = malloc(strlen(buffer)+1)) == NULL)
```

La place en mémoire est allouée dynamiquement par un appel à la fonction `malloc()`. La longueur de la chaîne est majorée de 1 pour tenir compte du terminateur. La fonction renvoie un pointeur qui est rangé en `lines[n]` s'il est différent de `NULL`. Cette dernière valeur indiquerait qu'il n'y a plus de mémoire disponible. On reviendrait alors au programme appelant avec une valeur égale à –1, ce dernier afficherait "Erreur d'allocation mémoire" à la ligne 25 et se terminerait immédiatement.

Si l'allocation de mémoire a réussi, le programme recopie la chaîne lue (pointée par `buffer`) dans la zone allouée en appelant la fonction de bibliothèque `strcpy()`. Ensuite, la boucle se répète.

Le lecteur attentif ne manquera pas de remarquer que, si l'allocation dynamique de mémoire est correctement effectuée, les auteurs ont malheureusement oublié de restituer cette mémoire (avec l'aide de la fonction `free()`) avant de mettre fin au programme. Le lecteur pourra admettre que cet oubli était destiné à éveiller sa sagacité.

Quant au tri, nous avons esquissé son algorithme plus haut et la fonction `sort()` ne fait que le mettre en application. La Figure 15.5 montre la façon dont se présentent les pointeurs avant le début du tri. On remarque que, lors de la première passe, si on a lu *n* lignes, on va faire *n – 1* comparaisons ; qu'on en fera *n – 2* lors de la seconde passe ; et ainsi de suite jusqu'à la fin, où la dernière ligne sera automatiquement en place. Les permutations de pointeurs vers les chaînes se font par les instructions des lignes 58 à 60. Si les deux chaînes sont égales, on les laisse en place.

À la fin du tri, les pointeurs peuvent se trouver (et ce sera généralement le cas, sauf si les phrases étaient déjà dans le bon ordre) bouleversés (voir Figure 15.6).

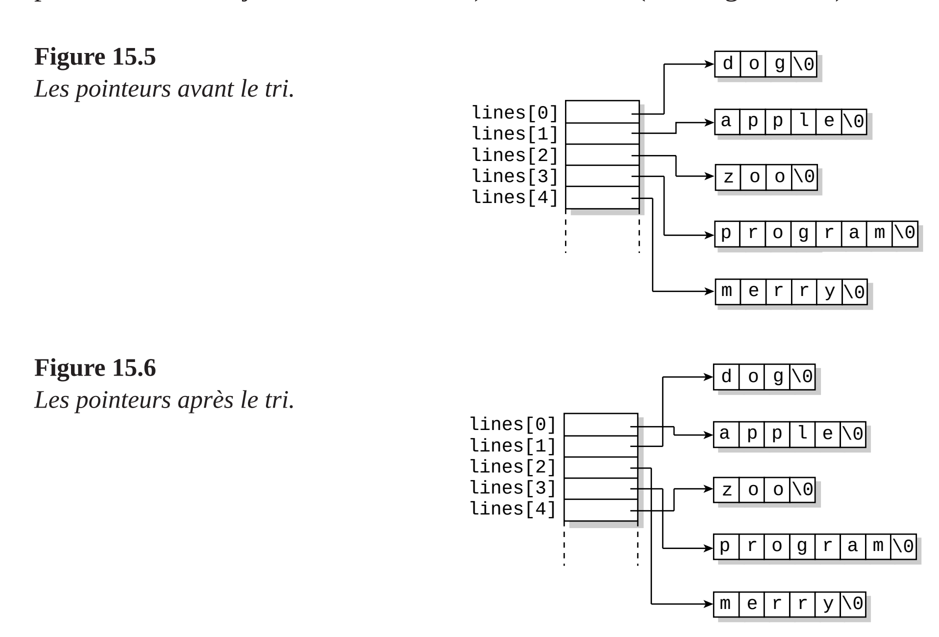


Figure 15.5
Les pointeurs avant le tri.

Figure 15.6
Les pointeurs après le tri.

Pointeurs vers des fonctions

Les pointeurs vers des fonctions offrent un autre moyen d'appeler des fonctions. Il y a là de quoi vous surprendre puisqu'un pointeur représente généralement l'adresse d'une variable. Cela n'est pas tout à fait exact. Il faudrait dire "l'adresse d'un objet C". Et une fonction EST un objet C. Elle se trouve quelque part en mémoire et son adresse est donc connue (sinon, comment ferait-on pour l'appeler ?).

Pourquoi peut-on avoir besoin d'utiliser un pointeur vers une fonction ? Tout simplement pour permettre d'appeler une "fonction va", c'est-à-dire une fonction ou une autre, selon tel ou tel critère. Le nom de la fonction à appeler est alors passé sous forme de pointeur vers la fonction choisie.

Déclaration d'un pointeur vers une fonction

Comme pour les variables, un pointeur vers une fonction doit être déclaré. La forme générale de ce type de déclaration est la suivante :

```
type (*ptr_vers_fonction)(liste_d_arguments);
```

Cela signifie qu'on déclare un pointeur appelé `ptr vers fonction` qui renvoie un résultat `type` et admet les arguments énumérés dans `liste d arguments`. Voici quelques exemples de déclarations :

```
int (*fonc1)(int x);
void (*fonc2)(double y, double z);
char (*fonc3)(char *p[]);
void (*fonc(4));
```

La première déclaration déclare `fonc1` comme étant un pointeur vers une fonction de type `int` acceptant un argument de type `int`. Dans la deuxième, `fonc2` est un pointeur vers une fonction de type `void` acceptant deux arguments de type `double`, tandis que pour la troisième, `fonc3` est un pointeur vers une fonction de type `char` acceptant un argument qui est un tableau de pointeurs de type `char`. Enfin, dans la dernière, `fonc4` est un pointeur vers une fonction de type `void` sans argument.

Le nom de la fonction doit être placé entre parenthèses pour des raisons liées à la priorité de l'opérateur d'indirection *. Oublier ces parenthèses conduirait à de sérieux problèmes.

Initialisation et utilisation d'un pointeur vers une fonction

Un pointeur vers une fonction doit naturellement être déclaré comme tout pointeur mais, en outre, il doit être initialisé afin de pointer vers une fonction. Les arguments et la valeur de retour de cette fonction doivent correspondre à ce qui a été déclaré dans le prototype de la déclaration du pointeur. Voici un exemple :

```
float carre(float z);   /* prototype de la fonction */
float (*p) (float x);   /* déclaration du pointeur */

float carre(float x);   /* la fonction elle-même */
{ return x * x;
}
```

On peut maintenant écrire, par exemple :

```
p = carre;
reponse = p(x);
```

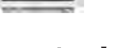
C’est simple mais, pour l’instant, on ne voit pas très bien à quoi ça peut servir. (Patience, le programme du Listing 15.9 éclairera votre lanterne !) Le Listing 15.8 en montre une application directe.

Listing 15.8 : Utilisation d’un pointeur vers une fonction pour appeler une fonction

```
/* Utilisation d'un pointeur vers une fonction pour appeler une
   fonction. */
#include <stdio.h>
#include <stdlib.h>
double square(double x);  /* prototype de la fonction. */
double (*p)(double x);    /* déclaration du pointeur. */

int main()
{ p = square;           /* p pointe vers square() */

 /* Appel de square() de deux façons. */
 printf("%f  %f\n", square(6.6), p(6.6));
 exit(EXIT_SUCCESS);
}
double square(double x)   /* la fonction square() elle-même. */
{ return x * x;
}
```



```
43.559999  43.559999
```

Analyse

Rien à ajouter : c’est la transcription exacte de l’exemple précédent.

Comme pour les pointeurs vers des variable numériques, rappelons qu’un nom de fonction sans parenthèses est un pointeur vers la fonction en question. L’intérêt d’utiliser un pointeur séparé est de pouvoir modifier son contenu ; donc la désignation de l’objet sur lequel il pointe, ce qui n’est pas possible autrement.

Le programme du Listing 15.9 appelle une fonction qui va elle-même appeler une fonction différente selon l’argument qui lui aura été passé. Chacune des trois fonctions possibles affiche un message particulier.

Listing 15.9 : Utilisation d’un pointeur vers une fonction pour choisir entre plusieurs fonctions à appeler

```
/* Utilisation d'un pointeur pour appeler différentes fonctions. */
#include <stdio.h>
#include <stdlib.h>

/* prototypes des fonctions. */
void func1(int x);
void one(void);
void two(void);
void other(void);

int main()
{ int a;

  for (;;)
  { puts("\nTapez un entier compris entre 1 et 10, 0 pour quitter");
    scanf("%d", &a);
    if (a == 0) break;
    func1(a);
  }
  exit(EXIT_SUCCESS);
}
void func1(int x)
{ void (*ptr)(void);

  if (x == 1) ptr = one;
   else if (x == 2) ptr = two;
    else ptr = other;

  ptr();
}

void one(void)
{ puts("Vous avez tapé 1");
}

void two(void)
{ puts("Vous avez tapé 2");
}

void other(void)
{ puts("Vous n'avez tapé ni 1 ni 2");
}
```



```
Tapez un entier compris entre 1 et 10, 0 pour quitter
1
Vous avez tapé 1

Tapez un entier compris entre 1 et 10, 0 pour quitter
2
```

```
Vous avez tapé 2

Tapez un entier compris entre 1 et 10, 0 pour quitter
3
Vous n'avez tapé ni 1 ni 2

Tapez un entier compris entre 1 et 10, 0 pour quitter
0
```

Analyse

La boucle infinie `for` de la ligne 14 lit une valeur tapée au clavier et appelle la fonction `func1()` si cette valeur n'est pas nulle (on se demande d'ailleurs pourquoi avoir demandé à l'utilisateur de limiter son choix entre 1 et 10). Si la valeur lue est nulle, le programme s'arrête.

Dans cette fonction, on commence par déclarer un pointeur vers une fonction (`ptr`). Selon la valeur passée à la fonction, ce pointeur est initialisé avec les noms `one`, `two` ou `other`. À la ligne 29, on appelle tout simplement `ptr()`. Selon la valeur passée en argument à `func1()`, cela permettra d'appeler une des trois fonctions `one()`, `two()` ou `other()`.

Le programme du Listing 15.10 est une version modifiée du précédent :

Listing 15.10 : Nouvelle version du programme du Listing 15.9

```
/* Passer un pointeur vers une fonction en argument */
#include <stdio.h>
#include <stdlib.h>

#define TOUJOURS 1

/* prototypes des fonctions */
void func1(void (*p)(void));
void one(void);
void two(void);
void other(void);

int main()
{ void (*ptr)(void);    /* pointeur vers une fonction */
  int  a;

  while(TOUJOURS)
  { puts("\nTapez un entier positif; 0 pour terminer.");
    scanf("%d", &a);

    if (a == 0) break;
    else if (a == 1)  ptr = one;
    else if (a == 2)  ptr = two;
    else ptr = other;

```

```
        func1(ptr);
      }
      exit(EXIT_SUCCESS);
    }
    void func1(void (*p)(void))
    { p();
    }

    void one(void)
    { puts("Vous avez tapé 1");
    }

    void two(void)
    { puts("Vous avez tapé 2");
    }

    void other(void)
    { puts("Vous n'avez tapé ni 1 ni 2");
    }
```

```
Tapez un entier positif; 0 pour terminer.
34
Vous n'avez tapé ni 1 ni 2

Tapez un entier positif; 0 pour terminer.
2
Vous avez tapé 2

Tapez un entier positif; 0 pour terminer.
1
Vous avez tapé 1

Tapez un entier positif; 0 pour terminer.
0
```

Analyse

Les différences avec le programme précédent sont minimes. D'abord, nous avons préféré une boucle `while` portant sur une expression différente de zéro (la constante `TOUJOURS`), ce qui est à la fois plus élégant et plus lisible.

Ensuite, ce n'est plus `func1()` qui effectue la sélection de la fonction à appeler mais, directement, la fonction `main()` qui va passer à `func1()` le pointeur vers la fonction à appeler qu'elle vient d'initialiser.

Enfin, nous avons supprimé la restriction du nombre à taper dont nous avions signalé l'inutilité dans la version précédente.

Revenons maintenant au programme du Listing 15.7. L'ordre de tri est déterminé par les valeurs de retour renvoyées par la fonction de comparaison appelée par `qsort()`. Celle-ci

exploite la fonction de bibliothèque `strcmp()`. Afin de pouvoir trier dans le sens ascendant ou dans le sens descendant, nous allons écrire deux fonctions de tri, l'une dans le sens normal (`alpha()`), l'autre dans le sens opposé (`reverse()`) :

```
/* Comparaison en ordre ascendant */
int alpha(char *p1, char *p2)
{ return(strcmp(p2, p1));
}

/* Comparaison en ordre descendant */
int reverse(char *p1, char *p2)
{ return(strcmp(p1, p2));
}
```

Remarquons l'astuce utilisée pour obtenir une sélection dans l'ordre alphabétique inverse : on s'est contenté de renverser l'ordre de comparaison et, au lieu de comparer la première chaîne à la seconde, on compare la seconde à la première.

Le choix entre ces deux fonctions sera effectué par l'utilisateur. On aboutit ainsi au programme du Listing 15.11 :

Listing 15.11 : Programme de tri dans un sens ou dans l'autre

```
/* Tri d'une suite de lignes de texte. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINES 25

int get_lines(char *lines[]);
void sort(char *p[], int n, int sort_type);
void print_strings(char *p[], int n);
int alpha(char *p1, char *p2);
int reverse(char *p1, char *p2);

char *lines[MAXLINES];

int main()
{ int number_of_lines, sort_type;

 /* Lire les lignes au clavier */

 number_of_lines = get_lines(lines);

 if (number_of_lines < 0)
 { puts("Erreur d'allocation mémoire");
   exit(EXIT_FAILURE);
 }
```

```

 printf("Tapez 0 pour trier en ordre alphabétique inverse,\n");
 printf("ou 1, pour trier en ordre alphabétique direct : ");
 scanf("%d", &sort_type);

 sort(lines, number_of_lines, sort_type);
 print_strings(lines, number_of_lines);
 exit(EXIT_SUCCESS);
}

int get_lines(char *lines[])
{int n = 0;
 char buffer[80]; /* Zone de lecture pour chaque ligne */

 puts("Tapez les lignes une par une ; une ligne vierge \
       pour terminer");
 while (n < MAXLINES && lire_clavier(buffer, sizeof(buffer)) != 0)
 { if ((lines[n] = malloc(strlen(buffer)+1)) == NULL)
     return -1;
   strcpy(lines[n++], buffer);
 }

 return n;

} /* Fin de get_lines() */

void sort(char *p[], int n, int sort_type)
{int a, b;
 char *x;

 int (*compare)(char *s1, char *s2);
 /* ptr vers fonction de comparaison */
 /* Initialiser le pointeur pour qu'il pointe sur la fonction
    de comparaison à appeler selon l'argument passé */

 compare = (sort_type) ? reverse : alpha;

 for (a = 1; a < n; a++)
  for (b = 0; b < n-1; b++)
  { if (compare(p[b], p[b+1]) > 0)
    { x = p[b];
      p[b] = p[b+1];
      p[b+1] = x;
    }
  }
} /* Fin de sort() */

void print_strings(char *p[], int n)
{ int count;

```

Listing 15.11 : Programme de tri dans un sens ou dans l'autre *(suite)*

```
    for (count = 0; count < n; count++)
     printf("%s\n ", p[count]);
}

int alpha(char *p1, char *p2) /* Comparaison en ordre ascendant */
{ return(strcmp(p2, p1));
}

int reverse(char *p1, char *p2) /*  Comparaison en ordre descendant */
{ return(strcmp(p1, p2));
}
```

```
Tapez les lignes une par une; une ligne vierge pour terminer
piano à bretelles
prunier
hortensia
abricotier
pommier
hortensia
trombone à coulisse

Tapez 0 pour trier en ordre alphabétique inverse,
ou 1, pour trier en ordre alphabétique direct :
0
trombone à coulisse
prunier
pommier
piano à bretelles
hortensia
hortensia
abricotier

Tapez les lignes une par une; une ligne vierge pour terminer
piano à bretelles
prunier
hortensia
abricotier
pommier
hortensia
trombone à coulisse

Tapez 0 pour trier en ordre alphabétique inverse,
ou 1, pour trier en ordre alphabétique direct :
1
abricotier
hortensia
hortensia
piano à bretelles
pommier
prunier
trombone à coulisse
```

Analyse

À la différence du Listing 15.7, un pointeur est déclaré vers la fonction de comparaison dans la fonction `sort()` :

```
int (*compare)(char *s1, char *s2);
```

Ce pointeur est ensuite initialisé de la façon suivante :

```
compare = (sort_type) ? reverse : alpha;
```

au moyen de l'opérateur ternaire (les parenthèses autour de `sort type` ne sont pas vraiment nécessaires). Il pointera donc soit vers `alpha()`, soit vers `reverse()`, selon la valeur de l'entier `sort type` passé en argument. Dans les deux boucles `for` de balayage du tableau de pointeurs, la comparaison de deux lignes s'effectuera ainsi :

```
if (compare(p[b], p[b+1]) > 0)
```

Ici, pas plus que dans le Listing 15.7, on ne libère les zones de mémoire allouées dynamiquement au tableau des pointeurs.



À faire

Mettre en pratique la programmation structurée.

Initialiser un pointeur avant de l'utiliser.

Libérer les zones de mémoire allouées dynamiquement !

À ne pas faire

Ne pas mettre des parenthèses aux bons endroits quand on déclare un pointeur vers une fonction. Voici deux exemples différents :

Déclaration d'un pointeur vers une fonction sans argument et qui retourne un caractère :

```
char (*fonc)();
```

Déclaration d'une fonction qui retourne un pointeur vers un caractère :

```
char *fonc();
```

Utiliser un pointeur vers une fonction qui aurait été déclaré avec un type de retour différent ou des arguments différents de ce que vous allez réellement utiliser.

Les listes chaînées

Une *liste chaînée* est une méthode d'enregistrement des données que l'on peut facilement implémenter avec le langage C. Nous traitons ce sujet dans un chapitre consacré aux pointeurs parce que ces derniers constituent un élément très important des listes chaînées.

Il existe plusieurs sortes de listes chaînées : les listes chaînées simples, les listes chaînées doubles, les arbres binaires... Chacun de ces types convient plus particulièrement à certaines tâches. Le point commun est que les liens entre les articles sont explicites et définis par des informations placées dans les articles eux-mêmes. C'est là une différence essentielle par rapport aux tableaux, dans lesquels les liens entre les articles sont implicites, et résultent de l'agencement général du tableau. Dans ce chapitre, nous allons expliquer les listes chaînées simples (que l'on appellera listes chaînées).

Principes de base des listes chaînées

Chaque élément de donnée d'une liste chaînée appartient à une structure (voir Chapitre 11). Celle-ci contient les éléments de données requis pour enregistrer les données spécifiques d'un programme. Cette structure contient un élément de donnée supplémentaire : un pointeur qui fournit les liens de la liste chaînée. Voici un exemple simple :

```
struct person {
char name[20];
struct person *next;
};
```

Ces lignes de code définissent la structure `person`. La partie données n'est constituée que d'un tableau de caractères de vingt éléments. L'utilisation d'une liste chaînée ne s'impose pas pour des données aussi simples, le seul intérêt de ces lignes est de constituer un exemple. La structure `person` contient aussi un pointeur de type `person`, il s'agit donc d'un pointeur vers une autre structure de même type. Cela signifie qu'une structure de type `person` contient un ensemble de données, et qu'elle peut aussi pointer vers une autre structure `person`. La Figure 15.7 présente le chaînage des structures dans une liste.

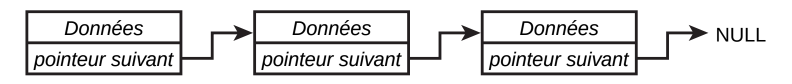


Figure 15.7
Établissement du chaînage entre les éléments d'une liste chaînée.

Remarquez que dans la Figure 15.7, chaque structure `person` pointe sur la structure `person` suivante. La dernière ne pointe sur rien. Pour concrétiser ce fait, on donne à son pointeur la valeur `NULL`.

Info

Les structures constituant une liste chaînée sont appelées liens, nœuds, ou éléments d'une liste chaînée.

Nous avons vu comment identifier le dernier élément d'une liste chaînée. Vous accédez au premier élément par l'intermédiaire d'un *pointeur de tête*, et cet élément contient un pointeur vers le deuxième. Ce deuxième élément contient un pointeur vers le troisième, et ainsi de suite jusqu'à ce que le pointeur rencontré ait la valeur `NULL`. Si la liste est vide (sans aucun lien), c'est le pointeur de tête qui a la valeur `NULL`. La Figure 15.8 présente ce pointeur de tête lors de l'initialisation de la liste, puis après l'ajout du premier élément.

Figure 15.8
Le pointeur de tête d'une liste chaînée.



Le pointeur de tête est un pointeur vers le premier élément d'une liste. On le désigne parfois sous le nom de "pointeur vers le sommet de la liste".

Utiliser les listes chaînées

Une liste chaînée, c'est un peu comme un fichier disque : des maillons peuvent être ajoutés, modifiés ou supprimés, mais il est indispensable d'ajuster les pointeurs de liaison (de chaînage) lorsqu'on exécute ce type d'opération.

Vous trouverez dans la suite de ce chapitre, un exemple de liste chaînée simple puis un programme plus complexe. Cependant, avant d'examiner le code, nous allons étudier quelques opérations indispensables à la manipulation des listes chaînées en utilisant la structure `person` présentée plus haut.

Préliminaire

Pour commencer une liste chaînée, vous devez définir une structure de données et le pointeur de tête. Ce pointeur doit être initialisé avec la valeur nulle puisque la liste est vide au moment de sa création. Vous aurez besoin d'un pointeur supplémentaire vers le type de structure de la liste pour pouvoir ajouter des enregistrements (vous verrez plus loin que des pointeurs supplémentaires pourront être nécessaires). Voici comment procéder :

```
struct person {
   char name[20];
```

```
    struct person *next;
};
struct person *new;
struct person *head;
head = NULL;
```

Ajouter le premier maillon

Si le pointeur de tête a la valeur `NULL`, la liste est vide et le nouveau maillon sera l'unique élément de la liste. Si ce pointeur a une valeur différente, la liste contient déjà un ou plusieurs éléments. Dans tous les cas, la procédure pour ajouter un maillon est identique :

1. Créez une structure en utilisant `malloc()` pour allouer la mémoire nécessaire.
2. Définissez le pointeur du nouvel élément avec la valeur du pointeur de tête. Cette valeur sera nulle si la liste est vide ou égale à l'adresse du premier élément en cours.
3. Modifiez la valeur du pointeur de tête avec l'adresse du nouvel élément.

Voici le code correspondant :

```
new = malloc(sizeof(struct person));
new->next = head;
head = new
```



Il est capital d'effectuer les opérations dans l'ordre indiqué, sinon, on perdrait le pointeur vers l'ancien premier maillon.

La Figure 15.9 présente l'ajout d'un maillon à une liste vide et la Figure 15.10 l'ajout du premier maillon à une liste existante.

La fonction `malloc()` permet d'allouer la mémoire pour le premier élément. On ne réserve en effet que la mémoire nécessaire à chaque création d'un élément supplémentaire. On aurait aussi bien pu faire appel à la fonction `calloc()`. Attention dans ce cas aux différences d'utilisation : `calloc()` initialisera le nouvel élément, pas `malloc()`.

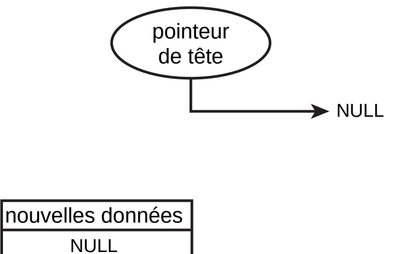


Dans l'exemple de code précédent, nous avons omis de tester le retour de `malloc()`. Vous ne devez pas suivre cet exemple, il faut toujours contrôler une allocation de mémoire.

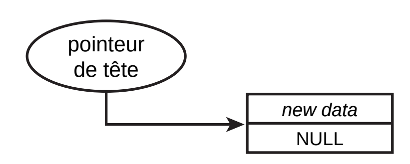


Quand on déclare un pointeur, il est bon de l'initialiser à `NULL` plutôt que de laisser sa valeur indéterminée.

Figure 15.9
Ajout d'un élément dans une liste vide.

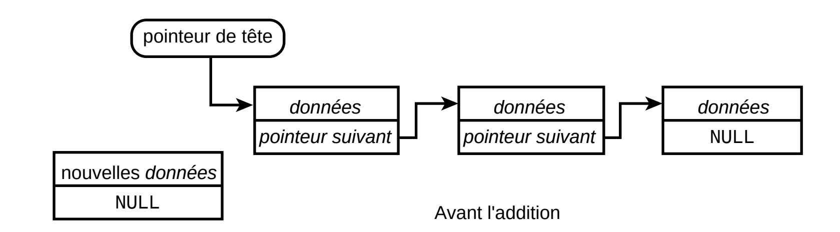


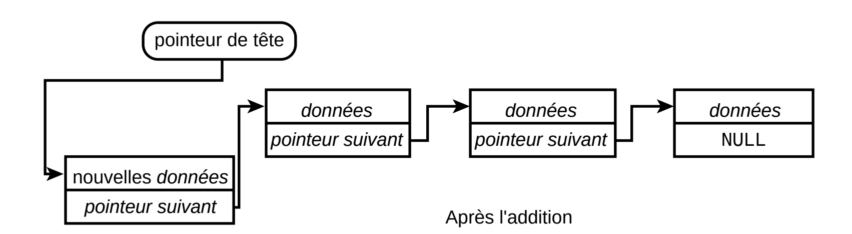
Avant l'addition



Après l'addition

Figure 15.10
Ajout dans une liste d'un nouveau premier élément.





Ajout d'un élément en queue de liste

À partir du pointeur de tête, vous devez parcourir la liste pour retrouver le dernier élément. Suivez ensuite ces étapes :

1. Créez une structure en utilisant `malloc()` pour allouer la mémoire nécessaire.
2. Redéfinissez le pointeur du dernier élément pour qu'il pointe vers le nouvel élément (dont l'adresse a été renvoyée par `malloc()`).
3. Définissez le pointeur du nouvel élément avec la valeur `NULL` qui indique la fin de la liste.

Voici le code correspondant :

```
person *current;
...
current = head;
while (current->next != NULL)
    current = current->next;
new = malloc(sizeof(struct person));
current->next = new;
new->next = NULL;
```

pointeur de tête
données
pointeur suivant
données
pointeur suivant
données
NULL
nouvelles données
NULL

Avant l'addition

pointeur de tête
données
pointeur suivant
données
pointeur suivant
données
pointeur suivant
nouvelles données
NULL

Après l'addition

Figure 15.11
Ajout d'un élément en queue de liste.

Ajout d'un maillon au milieu

Lorsque l'on travaille avec une liste chaînée, on doit la plupart du temps ajouter des maillons quelque part entre le premier et le dernier. L'emplacement d'insertion exact varie

en fonction de la gestion de la liste : elle peut être triée, par exemple, sur un ou plusieurs éléments de données. Vous devez donc d'abord vous placer au bon endroit de la liste avant d'effectuer l'ajout en suivant les étapes ci-après :

1. Localisez l'élément de la liste après lequel le nouveau maillon devra être inséré. Cet élément sera nommé élément de référence.
2. Créez une structure en utilisant `malloc()` pour allouer la mémoire nécessaire.
3. Modifiez le pointeur de l'élément de référence pour qu'il pointe vers le nouvel élément (dont l'adresse a été renvoyée par `malloc()`).
4. Définissez le pointeur du nouvel élément avec l'ancienne valeur de celui de l'élément de référence.

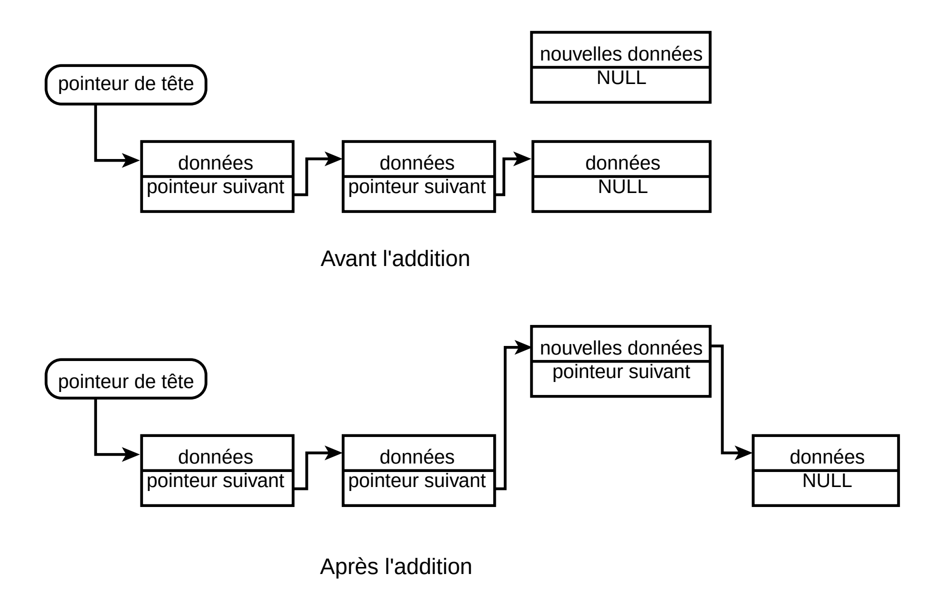


Figure 15.12
Ajout d'un élément au milieu d'une liste chaînée.

Voici le code correspondant :

```
person *marker;
/* Insérez ici le code nécessaire pour faire pointer
l'élément de référence (marker) sur l'emplacement requis
 de la liste.*/
...
new = malloc(sizeof(PERSON));
new->next = marker->next;
marker->next = new;
```

Suppression d’un élément de la liste

La suppression d’un maillon se résume à un simple ajustement des pointeurs. Le processus varie en fonction de l’emplacement de l’élément :

- Pour supprimer le premier élément, il faut faire pointer le pointeur de tête vers le deuxième élément.
- Pour supprimer le dernier élément, il faut donner au pointeur de l’avant dernier élément la valeur `NULL`.
- Pour supprimer un maillon intermédiaire, il faut modifier le pointeur de l’élément qui le précède pour le faire pointer vers l’élément qui suit l’élément à supprimer.

Les lignes de code suivantes suppriment le premier élément de la liste chaînée :

```
head = head->next;
```

Les lignes de code suivantes suppriment le dernier élément de la liste chaînée :

```
person *current1, *current2;
current1 = head;
current2= current1->next;
while (current2->next != NULL)
{
    current1 = current2;
    current2= current1->next;
}
current1->next = null;
if (head == current1)
    head = null;
```

Enfin, les lignes de code suivantes suppriment un élément intermédiaire de la liste chaînée :

```
person *current1, *current2;
/* Insérer ici le code nécessaire pour que current1 pointe vers
l'élément qui précède le maillon à supprimer. */
current2 = current1->next;
current1->next = current2->next;
```

Après l’exécution de ces lignes, l’élément supprimé est toujours en mémoire, mais il ne fait plus partie de la liste puisqu’il n’est plus "pointé". Dans vos programmes, n’oubliez pas de libérer la mémoire occupée par cet élément avec la fonction `free()` (cette fonction est traitée au Chapitre 20).

Un exemple de liste chaînée simple

Le Listing 15.12 illustre les opérations de base du traitement d'une liste chaînée. Il n'a pas d'autre objectif que celui de vous présenter le code correspondant à ces opérations puisqu'il ne reçoit aucune donnée de la part de l'utilisateur et qu'il ne réalise aucune tâche particulière :

1. Il définit une structure et les pointeurs destinés à la liste.
2. Il insère le premier élément de la liste.
3. Il ajoute un élément en fin de liste.
4. Il ajoute un élément en milieu de liste.
5. Il affiche la liste obtenue à l'écran.

Listing 15.12 : Les opérations de base dans une liste chaînée

```
 /* Illustre les opérations de base */
 /* dans une liste chaînée. */

 #include <stdlib.h>
 #include <stdio.h>
 #include <string.h>

 /* Structure d'un maillon. */
 struct data {
    char name[20];
    struct data *next;
    };

/* Définition des typedef de la structure */
/* et d'un pointeur vers celle-ci. */
typedef struct data PERSON;
typedef PERSON *LINK;

int main()
{
  /* Les pointeurs de tête (head), du nouvel élément (new),
  et de l'élément courant (current). */
  LINK head = NULL;
  LINK new = NULL;
  LINK current = NULL;
  /* Ajout du premier élément. Il ne faut jamais supposer */
  /* que la liste est vide au départ, même dans un */
  /* programme de démonstration comme celui-ci. */

  new = malloc(sizeof(PERSON));
  new->next = head;
  head = new;
  strcpy(new->name, "Abigail");

```

Listing 15.12 : Les opérations de base dans une liste chaînée *(suite)*

```
35:   /* Ajout d'un élément en fin de liste. */
36:   /* Nous supposons que la liste contient au moins */
37:   /* un élément. */
38:   current = head;
39:   while (current->next != NULL)
40:   {
41:     current = current->next;
42:   }
43:
44:   new = malloc(sizeof(PERSON));
45:   current->next = new;
46:   new->next = NULL;
47:   strcpy(new->name, "Catherine");
48:
49:   /* Ajoute un élément en seconde position dans la liste */
50:   new = malloc(sizeof(PERSON));
51:   new->next = head->next;
52:   head->next = new;
53:   strcpy(new->name, "Beatrice");
54:
55:   /* Affiche tous les maillons dans l'ordre. */
56:   current = head;
57:   while (current != NULL)
58:   {
59:     printf("%s\n", current->name);
60:     current = current->next;
61:   }
62:
63:   exit(EXIT_FAILURE);
64: }
```

On obtient l'affichage suivant :

```
Abigail
Beatrice
Catherine
```

Analyse

La structure de données de la liste est déclarée aux lignes 9 à 12. Les lignes 16 et 17 définissent les `typedef` de la structure et d'un pointeur vers cette structure. Le seul intérêt de ces lignes est de simplifier l'écriture de `struct data` en `PERSON` et de `struct data *` en `LINK`.

Les pointeurs qui permettront de manipuler la liste sont déclarés et initialisés en `NULL` aux lignes 22 à 24.

Les lignes 30 à 33 ajoutent un nouveau lien en tête de liste, et la ligne 30 alloue une nouvelle structure de données. Attention, nous supposons ici que la réservation de mémoire avec `malloc()` s'est déroulée avec succès. Vous devrez toujours contrôler le retour de cette fonction dans vos programmes.

La ligne 31 modifie le pointeur `next` de cette nouvelle structure pour le faire pointer à la même adresse que le pointeur de tête. Nous ne nous sommes pas contentés d'attribuer la valeur nulle à ce pointeur, car cette opération n'est valable qu'avec une liste vide. Rédigé de cette façon, ce code peut s'appliquer à une liste non vide. Le nouvel élément de tête va pointer sur l'élément qui était précédemment en tête, ce qui est bien le résultat recherché.

La ligne 32 fait pointer le pointeur de tête vers le nouvel enregistrement, et la ligne 33 y stocke quelques données.

L'ajout d'un élément en queue de liste est un peu plus compliqué. Notre liste ne contient qu'un élément, mais nous ne pouvons pas considérer ce cas particulier pour un programme normal. Il faut donc parcourir la liste à partir du premier élément jusqu'au dernier (la valeur du pointeur `next` est alors nulle). Cette opération est réalisée aux lignes 38 à 42. Il suffit ensuite d'allouer une nouvelle structure, de faire pointer le dernier élément précédent vers celle-ci et de donner la valeur nulle au pointeur `next` du nouvel élément (lignes 44 à 47).

La tâche suivante a permis d'ajouter un élément en milieu de liste : dans notre cas en deuxième position. Après l'allocation d'une nouvelle structure en ligne 50, le pointeur `next` du nouvel élément est défini pour pointer sur l'ancien deuxième élément qui s'est transformé en troisième élément (ligne 51), et le pointeur `next` du premier élément est défini pour pointer sur le nouvel élément (ligne 52).

Le programme se termine en affichant tous les maillons de la chaîne. Il commence avec l'élément sur lequel pointe le pointeur de tête, puis il progresse dans la liste jusqu'à trouver le dernier élément représenté par un pointeur `NULL` (lignes 56 à 61).

Implémentation d'une liste chaînée

Maintenant que nous avons vu chaque cas particulier, nous allons vous présenter, dans le programme du Listing 15.13, la réalisation d'une liste chaînée pouvant contenir cinq caractères. Il est entendu que ces caractères auraient pu être remplacés par n'importe quel autre type de données, le mécanisme étant le même. Pour simplifier, nous avons choisi un seul caractère.

Ici, nous devons trier les maillons lors d'un ajout. Cette opération supplémentaire qui donne au programme un caractère plus réaliste que le programme précédent. Chaque nouvel élément est ajouté à l'endroit approprié selon l'ordre naturel des caractères.

Listing 15.13 : Création d'une liste chaînée de caractères

```
 /*========================================================*
  * Program:  list1513.c                                   *
  * Objectif : implémenter une liste chaînée               *
  *========================================================*/

 #include <stdio.h>
 #include <stdlib.h>

/* Structure d'un maillon */
struct list
{
  int    ch; /*On utilise un int pour loger un caractère*/
  struct list *next_rec;
};

/* Les Typedef pour la structure et son pointeur. */
typedef struct list LIST;
typedef LIST *LISTPTR;

/* Prototypes des fonctions. */
LISTPTR add_to_list( int, LISTPTR );
void show_list(LISTPTR);
void free_memory_list(LISTPTR);

int main()
{
   LISTPTR first = NULL;  /* Pointeur de tête */
   int i = 0;
   int ch;
   char trash[256];   /* Pour effacer le buffer de stdin.*/

   while ( i++ < 5 ) /*Construire une liste de 5 éléments*/
   {
      ch = 0;
      printf("\nEntrez un caractère %d, ", i);

      do
      {
          printf("\nvaleurs entre a et z, svp: ");
          ch = getc(stdin);  /* lire le caractère suivant*/
          fgets(trash, sizeof(trash), stdin); /* nettoyer le buffer */
      } while( (ch < 'a' || ch > 'z')
      && (ch < 'A' || ch > 'Z'));
      first = add_to_list( ch, first );
   }

   show_list( first );    /* Afficher la liste en entier */
```

```
   free_memory_list( first );/*Restituer toute la mémoire*/
   exit(EXIT_FAILURE);
}

/*==========================================================*
 * Fonction: add_to_list()
 * But : Insérer un nouveau maillon dans la liste
 * Entrée   : int ch = caractère à insérer
 *           LISTPTR first = adresse du pointeur de tête
 * Retour : Adresse du pointeur de tête (first)
 *==========================================================*/

LISTPTR add_to_list( int ch, LISTPTR first )
{
   LISTPTR new_rec = NULL;   /* adresse du nouvel élément*/
   LISTPTR tmp_rec = NULL;   /* temporaire  */
   LISTPTR prev_rec = NULL;

   /* Allocation mémoire. */
   new_rec = malloc(sizeof(LIST));
   if (!new_rec)      /* Si plus de mémoire */
   {
      printf("Mémoire insuffisante!\n");
      exit(EXIT_FAILURE);
   }

   /* chaîner les données */
   new_rec->ch = ch;
   new_rec->next_rec = NULL;

   if (first == NULL) /*Ajout du premier maillon à la liste*/
   {
       first = new_rec;
       new_rec->next_rec = NULL;  /* redondant, mais sûr */
   }
   else    /* ce n'est pas le premier élément */
   {
      /* voir s'il doit précéder le premier élément */
      if ( new_rec->ch < first->ch)
      {
         new_rec->next_rec = first;
         first = new_rec;
      }
      else   /* il faut l'ajouter au milieu ou à la fin */
      {
         tmp_rec = first->next_rec;
         prev_rec = first;

         /* voir où on doit l'insérer. */

         if ( tmp_rec == NULL )
         {
```

Listing 15.13 : Création d'une liste chaînée de caractères *(suite)*

```
             /* ajout du second élément à la fin */
             prev_rec->next_rec = new_rec;
         }
         else
         {
            /* au milieu? */
            while (( tmp_rec->next_rec != NULL))
            {
               if( new_rec->ch < tmp_rec->ch )
               {
                  new_rec->next_rec = tmp_rec;
              if (new_rec->next_rec != prev_rec->next_rec)
                  {
                     printf("ERREUR");
                     getc(stdin);
                     exit(0);
                  }
                  prev_rec->next_rec = new_rec;
                  break; /* Le maillon a été ajouté, */
               }         /* fin du while             */
               else
               {
                 tmp_rec = tmp_rec->next_rec;
                 prev_rec = prev_rec->next_rec;
               }
            }

            /* voir si ajout à la fin */
            if (tmp_rec->next_rec == NULL)
            {
               if (new_rec->ch < tmp_rec->ch )
               {
                  new_rec->next_rec = tmp_rec;
                  prev_rec->next_rec = new_rec;
               }
               else  /* à la fin */
               {
                  tmp_rec->next_rec = new_rec;
                  new_rec->next_rec = NULL; /*Redondant */
               }
            }
         }
      }
   }
   return(first);
}

/*========================================================*
* Fonction: show_list
* But : Affiche le contenu de la liste chaînée
*========================================================*/

void show_list( LISTPTR first )
{
   LISTPTR cur_ptr;
   int counter = 1;
```

```

   printf("\n\nAdr élém    Position  Val.  Adr élém suivant\n");
   printf("==========  ========  ====  ================\n");

   cur_ptr = first;
   while (cur_ptr != NULL )
   {
      printf("  %p ", cur_ptr );
      printf(" %2i       %c", counter++, cur_ptr->ch);
      printf("      %p   \n",cur_ptr->next_rec);
      cur_ptr = cur_ptr->next_rec;
   }
}

/*========================================================*
* Fonction: free_memory_list
* But : libère la totalité de la mémoire acquise
*========================================================*/

void free_memory_list(LISTPTR first)
{
   LISTPTR cur_ptr, next_rec;
   cur_ptr = first;               /* Commencer au début */

   while (cur_ptr != NULL) /* jusqu'à la fin de la liste*/
   { next_rec = cur_ptr->next_rec; /*adresse élément */
                                   /* suivant        */
      free(cur_ptr);      /* libérer mémoire du maillon */
      cur_ptr = next_rec; /* ajuster pointeur courant*/
   }
}
```

L’exécution de ce programme donne le résultat suivant :

```
Entrez un caractère 1,
Valeur entre a et z, svp: q

Entrez un caractère 2,
Valeur entre a et z, svp: b

Entrez un caractère 3,
Valeur entre a et z, svp: z

Entrez un caractère 4,
Valeur entre a et z, svp: c

Entrez un caractère 5,
Valeur entre a et z, svp: a


Adr élém    Position  Val.  Adr élém suivant
==========  ========  ====  ================
0x8451C3A     1       a      0x8451C22
0x8451C22     2       b      0x8451C32
0x8451C32     3       c      0x8451C1A
0x8451C1A     4       q      0x8451C2A
0x8451C2A     5       z      0
```

Analyse

En analysant ce listing point par point, vous verrez qu'on y retrouve les différentes méthodes de mise à jour d'une liste que nous avons détaillées plus haut.



La meilleure façon de bien comprendre ce listing est d'exécuter le programme pas à pas à l'aide d'un débogueur. En surveillant l'évolution des divers pointeurs de chaînage, la mécanique de liaison devrait vous sembler plus claire.

Les déclarations initiales de début de programme devraient vous être familières. Les lignes 10 à 18 définissent la structure de la liste chaînée et les définitions de type qui vont simplifier l'écriture du code.

La fonction `main()` est simple, car elle fait appel à des fonctions pour la mise à jour de la liste chaînée. L'essentiel se trouve dans une boucle `while` à l'intérieur de laquelle se trouve une boucle `do...while`. Cette boucle intérieure s'assure que l'on a bien tapé un caractère alphabétique, minuscule ou majuscule, non accentué.

Lorsque le caractère est lu, on appelle la fonction `add to list()` à laquelle on transmet le pointeur de tête de liste et les données à ajouter.

`main()` se termine, après avoir lu cinq caractères, par un appel à `show list()` qui va afficher les caractères lus, dans l'ordre alphabétique, avec les pointeurs correspondants. La fin de l'affichage a lieu lorsque le pointeur vers l'élément suivant vaut `NULL`.

La fonction la plus importante, ici, est `add to list()` (lignes 52 à 145). C'est aussi la plus compliquée. On déclare d'abord (lignes 62 à 64) trois pointeurs qui seront utilisés pour pointer vers différents maillons.

`new rec` va pointer sur la mémoire qui a été allouée pour le nouvel élément (ligne 67). `new rec` pointera vers le nouveau maillon qui doit être ajouté. `tmp rec` pointera vers le maillon courant de la liste en cours d'évaluation. S'il y a plus d'un maillon dans la liste, `prev rec` sera utilisé pour pointer vers celui précédemment évalué. Remarquez qu'ici, nous testons le résultat de l'appel à `malloc()` et que, s'il est infructueux, un message est affiché, et le programme se termine (lignes 70 et 71).

À la ligne 75, on place le nouvel élément dans la structure pointée par `new rec`. Dans un programme réel, il y aurait généralement plusieurs objets à ranger ici. À la ligne suivante, le pointeur vers l'élément suivant est mis à `NULL`.

À la ligne 78, commence l'addition du maillon en regardant s'il y a déjà des éléments dans la liste. Si l'élément qu'on veut ajouter est le premier, on se contente de faire pointer le pointeur de tête, `first`, vers le nouvel élément (ligne 68).

Si le nouveau maillon n'est pas le premier, la fonction continue et on exécute la branche `else` qui commence à la ligne 83. La ligne 86 regarde si le nouveau maillon ne doit pas être placé en tête de liste. Si c'est le cas, les lignes 88 et 89 font le nécessaire.

Si les deux tests précédents ont échoué, on sait qu'il faut ajouter le nouveau maillon quelque part au milieu de la liste. Les lignes 93 et 94 ajustent les pointeurs `tmp rec` et `prev rec` définis plus haut. Le premier pointe vers l'adresse du second maillon de la liste, et `prev rec` prend la valeur du premier pointeur de la liste.

Vous remarquerez que, s'il n'y a qu'un seul élément dans la liste, `tmp rec` vaut `NULL`. Ce cas particulier est testé à la ligne 98. Si c'est le cas, on sait que le nouveau maillon sera le second, c'est-à-dire à la fin de la liste. Pour cela, on se contente de faire pointer `prev rec >next ptr` vers le nouveau maillon et le tour est joué.

Si `tmp rec` ne vaut pas `NULL`, on sait qu'il y a plus de deux maillons dans la liste. `while`, des lignes 106 à 125, va boucler sur le reste des maillons pour savoir à quelle place on doit insérer le nouveau maillon. À la ligne 108, on regarde s'il est inférieur à celui qui est actuellement pointé. Si c'est le cas, on sait que c'est là qu'il faut l'insérer. Si le nouveau maillon est supérieur au maillon courant, il faut le comparer au suivant. Aux lignes 122 et 123, on incrémente `tmp rec` et `next rec`.

Si le caractère à insérer (c'est-à-dire le nouveau maillon) est inférieur à celui du maillon courant, on suit la logique présentée plus haut pour l'ajouter en milieu de liste : lignes 110 à 118. Une fois que c'est fait, la boucle `break` permet de quitter `while`.



Les lignes 111 à 116 contiennent du code de mise au point qui a été laissé dans le listing pour vous montrer comment tester ce genre de construction logicielle. Une fois que le programme tourne correctement, on doit normalement enlever ces instructions.

L'ajout en fin de liste a déjà été traité plus haut. Il se traduit par une sortie normale de la boucle `while` (lignes 106 à 125). L'insertion s'effectue au moyen des instructions présentes aux lignes 128 à 140.

Quand on atteint le dernier maillon, `tmp rec >next rec` doit être égal à `NULL`. C'est ce qui est testé en ligne 128. À la ligne 130, on regarde si le maillon doit être inséré avant ou après le dernier. S'il doit être placé après, on fait pointer `next rec` sur le nouveau maillon et celui-ci sur `NULL` (ligne 138).

Développements du programme du Listing 15.13

Les listes chaînées ne sont pas aussi simple à maîtriser. L'exemple que nous venons de voir montre qu'on peut, grâce à elles, construire des listes ordonnées. Au lieu d'un seul caractère, on aurait pu traiter des chaînes de caractères, des numéros de téléphone ou n'importe quelle autre information. Ici, on avait choisi l'ordre croissant, mais on aurait pu adopter l'ordre décroissant.

Suppression dans une liste chaînée

Pour être complet, il faudrait aussi pouvoir réaliser la suppression d'un maillon de la liste. L'idée générale est la même, et il ne faut pas oublier de libérer la mémoire correspondant aux maillons supprimés.



À faire

Essayer de bien comprendre la différence entre `calloc()` *et* `malloc()`. *En particulier, souvenez-vous que* `calloc()` *initialise la mémoire allouée, ce que ne fait pas* `malloc()`.

À ne pas faire

Oublier de libérer la mémoire correspondant aux maillons supprimés.

Résumé

Dans ce chapitre, nous avons passé en revue plusieurs façons évoluées d'utiliser des pointeurs. Vous savez, maintenant, que les pointeurs constituent l'essence même du C. Vous verrez, à l'usage, que rares sont les programmes C qui n'en font pas usage. Vous avez vu comment utiliser des pointeurs pointant vers d'autres pointeurs, et à quoi peuvent servir des tableaux de pointeurs. Vous avez découvert comment C traite les tableaux à plusieurs dimensions qui sont des tableaux de tableaux, et vous avez vu comment employer les pointeurs pour ces tableaux. Vous avez étudié l'utilisation des pointeurs vers des fonctions. Finalement, vous avez appris à implémenter les listes chaînées, une méthode très efficace d'enregistrement des données.

Ce chapitre a été quelque peu ardu mais, bien que ces sujets soient un peu compliqués, leur intérêt est certain. Vous touchez là à quelques-unes des possibilités les plus sophistiquées du C qui justifient sa popularité actuelle.

Q & R

Q Quelle est la profondeur maximale qu'on peut utiliser avec des pointeurs pointant vers des pointeurs ?

R Il n'y a pas de limite théorique. Tout dépend d'éventuelles restrictions propres au compilateur que vous utilisez. Il est rare, cependant, d'avoir besoin de dépasser une profondeur de 3.

Q Y a-t-il une différence entre un pointeur vers une chaîne de caractères et un pointeur vers un tableau de caractères ?

R Non. C'est la même chose.

Q Est-il nécessaire de mettre en pratique les concepts qui viennent d'être présentés dans ce chapitre pour tirer tous les avantages possibles du C ?

R Pas vraiment. Mais vous ne profiterez pas de tout ce qui fait la puissance de ce langage.

Q Y a-t-il d'autres circonstances dans lesquelles on peut être amené à utiliser des pointeurs vers des fonctions ?

R Oui. Avec des menus, par exemple.

Q Quels sont les deux avantages importants des listes chaînées ?

R Le premier est que la taille d'une liste chaînée évolue pendant l'exécution du programme, vous n'avez pas besoin de la prévoir en créant le code. Le second est que ce genre de liste peut facilement être triée, puisque l'on ajoute ou que l'on supprime des maillons n'importe où.

Atelier

L'atelier vous propose quelques questions permettant de tester vos connaissances sur les sujets que nous venons d'aborder dans ce chapitre.

Quiz

1. Écrivez un processus qui déclare une variable `x` de type `float`, déclare et initialise un pointeur vers la variable, et déclare et initialise un pointeur vers ce pointeur.

2. Continuez l'exemple ci-avant. Supposez que vous vouliez utiliser le pointeur vers un pointeur pour assigner la valeur `100` à la variable `x`. Qu'y a-t-il éventuellement de faux dans l'instruction suivante ?

```
*ppx = 100;
```

3. Supposons que vous ayez déclaré un tableau de la façon suivante :

```
int tableau[2][3][4];
```

   Quelle est la structure de ce tableau, vue par le compilateur C ?

4. Avec ce même tableau, que signifie l'expression `tableau[0][0]` ?

5. Toujours avec ce même tableau, quelles sont, parmi ces trois comparaisons, celles qui répondent `TRUE` ?

```
tableau[0][0] == &tableau[0][0][0]
tableau[0][1] == tableau[0][0][1]
tableau[0][1] == &tableau[0][1][0]
```

6. Écrivez le prototype d'une fonction qui accepte un tableau de pointeurs de type `char` comme argument et ne renvoie rien.
7. Comment la fonction de la question 6 sait-elle combien d'éléments il y a dans le tableau de pointeurs qui lui a été passé ?
8. Qu'est-ce qu'un pointeur vers une fonction ?
9. Écrivez la déclaration d'un pointeur vers une fonction qui renvoie un `char` et accepte un tableau de pointeurs vers un `char` comme argument.
10. Si vous aviez répondu à la question 9 par :

```
char *ptr(char *x[]);
```

qu'y aurait-il de faux ?

11. Lorsque vous définissez la structure destinée à une liste chaînée, quel élément devez-vous inclure ?
12. Que signifie une valeur `NULL` pour un pointeur de tête ?
13. Comment sont reliés les éléments d'une liste chaînée ?
14. Que font les déclarations suivantes ?
    a) `int *var1;`.
    b) `int var2;`.
    c) `int **var3;`.
15. Que font les déclarations suivantes ?
    a) `int a[3][12];`.
    b) `int (*b)[12];`.
    c) `int *c[12];`.
16. Que font les déclarations suivantes ?
    a) `char *z[10];`.
    b) `char *y(int champ);`.
    c) `char (*x)(int champ);`.

Exercices

1. Écrivez une déclaration pour un pointeur vers une fonction qui accepte un entier comme argument et renvoie une variable de type `float`.
2. Écrivez une déclaration pour un tableau de pointeurs vers des fonctions. Les fonctions accepteront toutes une chaîne de caractères comme argument et renverront un entier. À quoi pourrait servir un tel tableau ?
3. Écrivez une déclaration convenant à un tableau de dix pointeurs de type `char`.
4. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions suivantes ?

   ```
   int x[3][12];
   int *ptr[12];
   ptr = x;
   ```

5. Créez une structure qui sera utilisée avec une liste simplement chaînée. Elle devra recevoir les noms et adresses de vos connaissances.

   Comme plusieurs solutions sont possibles, nous ne donnons pas les réponses aux trois questions suivantes.

6. Écrivez un programme qui déclare un tableau de 12 x 12 caractères. Placez un X dans chaque élément et affichez le résultat sous forme de grille en utilisant un pointeur vers le tableau.
7. Écrivez un programme qui range dix pointeurs vers des variables de type `double`. Le programme devra accepter dix nombres tapés par l'utilisateur, les trier et les afficher (consultez éventuellement le Listing 15.10).

16

Utilisation de fichiers sur disque

Parmi les programmes que vous écrivez, beaucoup utilisent des fichiers sur disque pour diverses tâches : sauvegarde de données ou d'informations de configuration, par exemple. Voici ce que vous allez étudier dans le présent chapitre :

- Établissement d'un lien entre les flots et les fichiers sur disque
- Deux types de fichiers sur disque C
- Ouverture d'un fichier
- Écriture de données dans un fichier
- Lecture de données à partir d'un fichier
- Fermeture d'un fichier
- Gestion des fichiers sur disque
- Utilisation de fichiers temporaires

Flots et fichiers sur disque

Comme vous l'avez appris au Chapitre 14, C réalise ses entrées/sorties au moyen de *flots*. Vous savez comment utiliser les flots prédéfinis du C, correspondant à des périphériques spécifiques tels que le clavier, l'écran. Les flots des fichiers sur disque opèrent de façon identique. C'est un des avantages des entrées/sorties par flots. La principale nouveauté avec les flots des fichiers sur disque est qu'ici, c'est votre programme qui doit explicitement créer le flot associé à un fichier sur disque particulier.

Types de fichiers sur disque

Au Chapitre 14, vous avez vu qu'il existait deux sortes de flots : *texte* et *binaire*. Vous pouvez associer chaque type de flot à un fichier, et il est important de bien comprendre la distinction entre ces deux types pour les utiliser à bon escient.

Un *flot de type texte* est associé à un fichier en mode texte. Chaque ligne contient un nombre de caractères compris entre 0 et 255 (bornes comprises), et qui se termine par un ou plusieurs caractères signifiant *fin de ligne*. Une ligne n'est pas une chaîne de caractères ; il n'y a pas de `\0` terminal. Lorsque vous utilisez un fichier en mode texte, une traduction s'effectue entre le caractère de fin de ligne du C, `\n`, et les caractères utilisés par le système d'exploitation comme terminateur de ligne. Sur les systèmes issus de DOS comme Windows, c'est une association <retour chariot>, <à la ligne>. Lorsqu'on écrit des informations vers un fichier sur disque, chaque `\n` est traduit par une combinaison <retour chariot>, <à la ligne>. Inversement, à la lecture, ces deux caractères sont traduits par un `\n`. Sur les systèmes UNIX, il n'y a aucune traduction, les caractères `\n` restent inchangés.

Tout ce qui n'est pas un fichier texte est un *flot de type binaire*. Ces flots sont associés avec des fichiers en mode binaire. Toutes les informations sont écrites et lues telles quelles, sans aucune séparation entre les lignes et sans caractères de fin de ligne. Les caractères `\0` et `\n` n'ont plus de signification particulière.

Certaines fonctions d'entrées/sorties sont limitées à un seul des deux types de fichiers, alors que d'autres peuvent utiliser les deux modes. Dans ce chapitre, nous allons étudier les fonctions associées aux deux modes.

Noms de fichiers

Pour travailler avec des fichiers sur disque, ceux-ci doivent avoir un nom. Les noms de fichier sont exprimés sous forme de chaînes de caractères, comme n'importe quel texte. Les noms sont ceux qu'utilise le système d'exploitation ; ils doivent suivre les mêmes règles.

Les différents systèmes d'exploitation n'appliquent pas forcément les même règles pour l'autorisation des caractères dans les noms de fichiers. Les caractères suivants, par exemple, sont interdits en Windows :

```
/ \ : * ? " < > |
```

Dans un programme C, un nom de fichier peut aussi contenir des informations concernant le chemin d'accès. Ce chemin représente l'unité et/ou le répertoire dans lequel se situe ce fichier. Si votre nom de fichier n'indique pas le chemin d'accès, on suppose que ce fichier se situe à l'emplacement courant par défaut du système d'exploitation. L'indication de ce chemin est cependant une bonne habitude à prendre en programmation.

Sur Windows, l'antislash (`\`) sépare les noms de répertoire dans le chemin d'accès. Par exemple, le nom

```
c:\data\liste.txt
```

se réfère à un fichier, appelé liste.txt, situé dans le répertoire \data du disque C. Vous n'ignorez pas que l'antislash (`\`) prend un sens particulier en C, lorsqu'il apparaît dans une chaîne de caractères. Pour représenter ce caractère lui-même, on doit le redoubler. Ainsi, dans un programme C, une chaîne de caractères représentant un nom de fichier complet s'écrit :

```
char *nomfich="c:\\data\\liste.txt";
```

Si vous tapez un nom de fichier au clavier, vous ne tapez, bien entendu, qu'un seul anti-slash.

Certains systèmes d'exploitation utilisent d'autres séparateurs pour les noms de répertoires. UNIX et Linux, par exemple, utilisent le slash normal (`/`).

Ouverture d'un fichier

Le processus permettant d'établir un lien entre un flot et un fichier sur disque est appelé *l'ouverture* d'un fichier. Lorsque vous ouvrez un fichier, il devient utilisable en lecture (les données peuvent être entrées dans le programme), en écriture (le programme peut envoyer des résultats dans le fichier) ou dans les deux modes à la fois. Lorsque vous avez fini d'utiliser un fichier, vous devez le refermer. Ce point sera abordé plus loin, dans ce chapitre.

Pour ouvrir un fichier, utilisez la fonction de bibliothèque `fopen()`. Le prototype de `fopen()` est situé dans stdio.h, où il apparaît ainsi :

```
FILE *fopen(const char *filename, const char *mode);
```

Ce prototype vous indique que `fopen()` renvoie un pointeur de type `FILE` qui est une structure déclarée dans stdio.h. Les membres de cette structure sont utilisés par le programme pour diverses opérations d'accès au fichier, mais il est inutile de vous en préoccuper. Ce que vous devez retenir, c'est que, pour chaque fichier que vous voulez ouvrir, vous devez déclarer un pointeur de type `FILE`. Lorsque vous appelez `fopen()`, cette fonction crée une instance de la structure `FILE` et renvoie un pointeur vers cette structure. Vous utiliserez ce pointeur dans les opérations ultérieures sur ce fichier. Si la fonction `fopen()` échoue, elle renvoie `NULL`. Elle peut échouer, par exemple, à cause d'une erreur matérielle ou d'une tentative d'ouverture d'un fichier inexistant.

L'argument `filename` est le nom du fichier à ouvrir. Comme nous l'avons dit plus haut, il peut contenir le nom du disque et/ou le chemin d'accès. Il peut être représenté par une chaîne de caractères constante, placée entre guillemets, ou par un pointeur vers une chaîne de caractères située n'importe où en mémoire.

L'argument `mode` spécifie le mode dans lequel on doit ouvrir le fichier. Dans ce contexte, `mode` commande le *mode* d'ouverture du fichier : texte ou binaire ; lecture ou écriture ou les deux. Les valeurs possibles de `mode` sont données par le Tableau 16.1.

Tableau 16.1 : Valeurs de mode pour la fonction fopen()

Mode	***Signification***
`r`	Ouverture du fichier en lecture. Si le fichier n'existe pas, `fopen()` renvoie `NULL`.
`w`	Ouverture du fichier en écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, son contenu est effacé.
`a`	Ouverture du fichier en mise à jour (*append*). Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont ajoutées à la fin.
`r+`	Ouverture du fichier en lecture et en écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont écrites en tête, écrasant celles qui s'y trouvaient précédemment.
`w+`	Ouverture du fichier en lecture et en écriture. Si le fichier n'existe pas, il est créé. S'il existe déjà, son contenu est écrasé.
`a+`	Ouverture du fichier en lecture et en mise à jour. Si le fichier n'existe pas, il est créé. S'il existe déjà, les nouvelles informations sont ajoutées à la fin.

Le mode par défaut est le mode texte. Pour ouvrir un fichier en mode binaire, vous devez ajouter un `b` à l'argument `mode`. Sur Linux, cela est facultatif car l'ouverture d'un fichier est systématiquement traitée en mode binaire. Un argument `mode` contenant `a` opérera une ouverture pour mise à jour en mode texte, alors que `ab` effectuera une ouverture pour mise à jour en mode binaire.

Souvenez-vous que `fopen()` renvoie `NULL` si une erreur survient. Les conditions d'erreur qui peuvent provoquer ce type d'incident sont :

- utilisation d'un nom de fichier non valide ;
- tentative d'ouverture d'un fichier dans un répertoire ou un disque inexistants ;
- tentative d'ouverture en mode lecture (`r`) d'un fichier inexistant.

Lorsque vous exécutez `fopen()`, vous devez toujours tester l'éventualité d'une erreur. Vous ne pouvez pas directement connaître le type de l'erreur, mais vous pouvez envoyer un message à l'utilisateur et essayer d'ouvrir à nouveau le fichier ou mettre fin au programme. La plupart des compilateurs C proposent des fonctions non ANSI/ISO avec lesquelles vous pouvez obtenir des informations sur la nature de l'erreur. Consultez la documentation de votre compilateur.

Le Listing 16.1 vous présente un programme de démonstration de la fonction `fopen()`.

Listing 16.1 : Utilisation de fopen() pour ouvrir un fichier sur disque en différents modes

```
/* Démonstration de la fonction fopen(). */
#include <stdlib.h>
#include <stdio.h>

int main()
{
    FILE *fp;
    char filename[40], mode[4];

    while (1)
    {

        /* Indiquer le nom de fichier et le mode. */

        printf("\nTapez un nom de fichier : ");
        lire_clavier(filename, sizeof(filename));
        printf("\nTapez un mode (3 caractères au plus) : ");
        lire_clavier(mode, sizeof(mode));

        /* Essayer d'ouvrir le fichier. */

        if ((fp = fopen(filename, mode)) != NULL)
        {
            printf("\nOuverture réussie %s en mode %s.\n",
                     filename, mode);
            fclose(fp);
            puts("Tapez x pour terminer, \
                 ou n'importe quoi d'autre pour continuer.");
            if (getc(stdin) == 'x')
```

Listing 16.1 : Utilisation de fopen() pour ouvrir un fichier sur disque en différents modes *(suite)*

```
29:                 break;
30:             else
31:                 continue;
32:         }
33:         else
34:         {   fprintf(stderr, "\nErreur à l'ouverture du fichier \
35:                              %s en mode %s.", filename, mode);
36:             puts("Tapez x pour terminer, \
37:                   ou n'importe quoi d'autre pour réessayer.");
38:             if (getc(stdin) == 'x')
39:                 break;
40:             else
41:                 continue;
42:         }
43:     }
44:     exit(EXIT_SUCCESS);
45: }
```

```
Tapez un nom de fichier : list1601.c
Tapez un mode (3 caractères au plus) : w
Ouverture réussie 16p1.c en mode w.
Tapez x pour terminer, ou n'importe quoi d'autre pour continuer.
j
Tapez un nom de fichier : abcdef
Tapez un mode (3 caractères au plus) : r
Erreur à l'ouverture du fichier abcdef en mode r.
Tapez x pour terminer, ou n'importe quoi d'autre pour réessayer.
x
```

Analyse

Le programme vous demande de lui indiquer un nom de fichier et un mode de lecture. Ensuite, `fopen()` essaie d'ouvrir le fichier en rangeant dans `fp` le pointeur vers le fichier (ligne 22). L'instruction `if` située sur la même ligne vérifie que tout s'est bien passé en comparant le pointeur renvoyé à `NULL`. Si `fp` ne vaut pas `NULL`, un message avertit l'utilisateur que tout va bien et qu'il peut continuer. Dans le cas contraire, c'est le bloc d'instructions placé après le `else` qui est exécuté : un message d'erreur est affiché expliquant qu'il y a eu un problème. On propose ensuite à l'utilisateur de continuer ou de terminer le programme.

Vous pouvez faire quelques essais avec divers noms de fichiers et différents modes pour étudier le comportement de ce programme. La façon la plus simple de produire une erreur est de demander l'ouverture en lecture d'un fichier qui n'existe pas (comme dans le cas du fichier `abcdef` de notre exemple).

Écriture et lecture d'un fichier de données

Un programme utilisant un fichier sur disque peut écrire des données dans un fichier, les lire ou faire les deux. Il y a trois façons d'écrire dans un fichier :

- Vous pouvez utiliser des sorties formatées pour sauvegarder des données mises en forme. Ce n'est valable que pour des fichiers texte. Les sorties formatées sont principalement utilisées lors de la création de fichiers contenant du texte et des valeurs numériques destinés à être lus par d'autres programmes tels que des tableurs ou des bases de données.
- Vous pouvez effectuer des sorties caractère par caractère pour écrire des caractères isolés ou des chaînes de caractères dans un fichier. Ce genre de pratique est maintenant rare car cela ralentit le programme de manière significative avec les systèmes d'exploitations modernes (Windows, Linux, *BSD...).
- Vous pouvez utiliser des sorties directes pour sauvegarder le contenu d'un bloc de mémoire directement vers un fichier sur disque. Cette méthode n'est valable que pour des fichiers binaires. L'utilisation de ce type de sortie est le meilleur moyen de sauvegarder des valeurs destinées à être réutilisées plus tard par un programme C sur une machine de même type.

Lorsque vous voulez lire des informations à partir d'un fichier, vous avez le choix entre ces trois mêmes options : entrée formatée, entrée caractère par caractère ou entrée directe. Le choix dépend presque entièrement de la nature du fichier à lire. Vous choisirez la plupart du temps de lire des données dans le mode avec lequel elles auront été sauvegardées, mais rien ne vous y oblige. La lecture d'un fichier dans un mode différent de celui dans lequel il a été écrit nécessite en effet une très bonne connaissance du C et des formats de fichier.

La description que nous venons de faire suggère qu'il existe des tâches bien appropriées pour chacun de ces trois types. Mais ce n'est pas une règle absolue. Le langage C est assez souple (c'est précisément l'un de ses avantages) pour qu'un programmeur adroit puisse utiliser n'importe quel type de fichier pour n'importe quelle application. Mais, si vous êtes un programmeur débutant, ne cherchez pas à vous compliquer la vie et suivez les conseils qui viennent d'être donnés.

Entrées et sorties formatées

Les entrées/sorties formatées concernent le texte et les données numériques formatés de façon particulière. On peut les comparer à ce qui se passe avec le clavier et l'écran lorsqu'on utilise les instructions `scanf()` et `printf()` décrites au Chapitre 14. Nous allons commencer par les sorties formatées.

Sorties formatées vers un fichier

Une sortie formatée vers un fichier se réalise en appelant la fonction de bibliothèque `fprintf()`. Le prototype de `fprintf()` se trouve dans le fichier d'en-tête stdio.h et se présente ainsi :

```
int fprintf(FILE *fp, char *fmt,...);
```

Le premier argument est un pointeur vers un objet de type `FILE`. Pour écrire des informations vers un fichier sur disque particulier, il faut passer à la fonction le pointeur récupéré au moment de l'ouverture de ce fichier par `fopen()`.

Le second argument est une chaîne de caractères contenant le format à utiliser. Vous avez déjà rencontré ce type de chaîne de caractères, lorsque nous avons étudié `printf()` au Chapitre 14. Nous retrouvons ici exactement le même type de chaîne. Pour plus de détails, vous pouvez donc vous reporter au Chapitre 14.

L'argument final est `...` Qu'est-ce que cela signifie ? Dans un prototype de fonction, des points de suspension (parfois appelés *ellipse*) représentent un nombre variable d'arguments. Autrement dit, outre le pointeur de fichier et la chaîne de caractères du format, `fprintf()` accepte zéro, un ou plusieurs arguments supplémentaires, exactement comme `printf()`. Ces arguments sont les noms des variables à écrire dans le flot spécifié.

Souvenez-vous que `fprintf()` fonctionne comme `printf()`, à ce détail près que les sorties sont envoyées vers un fichier sur disque, et non vers l'imprimante. Si vous indiquez comme flot de sortie `stdout`, les deux instructions se comporteront exactement de la même façon.

Le programme du Listing 16.2 utilise `fprintf()`.

Listing 16.2 : Démonstration de l'équivalence entre les sorties faites avec fprintf() vers un fichier sur disque et vers stdout

```
/* Démonstration de la fonction fprintf(). */
#include <stdlib.h>
#include <stdio.h>

void clear_kb(void);

int main()
{
    FILE *fp;
    float data[5];
    int count;
    char filename[20];

    puts("Tapez 5 valeurs numériques en flottant.");

```

```
    for (count = 0; count < 5; count++)
          scanf("%f", &data[count]);

    /* Demandez un nom de fichier et ouvrez le fichier.
       Commencez par vider stdin de tout caractère qui
       pourrait s'y trouver.
    */

    clear_kb();

    puts("Indiquez un nom pour le fichier.");
    lire_clavier(filename, sizeof(filename));

    if ((fp = fopen(filename, "w")) == NULL)
    {   fprintf(stderr, "Erreur à l'ouverture du fichier %s.",
                filename);
        exit(EXIT_FAILURE);
    }

    /* Écrivez les données numériques vers le fichier et stdout */

    for (count = 0; count < 5; count++)
    {
        fprintf(fp, "data[%d] = %f\n", count, data[count]);
        fprintf(stdout, "data[%d] = %f\n", count, data[count]);
    }
    fclose(fp);
    exit(EXIT_SUCCESS);
}

void clear_kb(void)
/* Vide stdin de tout caractère en attente. */
{
    char junk[80];
    fgets(junk, sizeof(junk), stdin);
}
```

```
Tapez 5 valeurs numériques en flottant.
123.4
-555.666
3.141592
-0.00876
123456.
Indiquez un nom pour le fichier.
cocorico.txt

data[0] = 123.400002
data[1] = -555.599976
data[2] = 3.141592
data[3] = -0.008760
data[4] = 123456.000000
```

Analyse

Vous remarquerez quelques différences entre les valeurs que vous avez tapées et celles qui seront affichées. Ce n'est pas une erreur dans le programme, mais la conséquence normale de la façon dont les nombres sont conservés dans la machine. Ce sont des erreurs de conversion entre la représentation externe des nombres et leur représentation interne.

Le programme utilise `fprintf()` deux fois de suite : la première fois vers le fichier sur disque dont l'utilisateur a donné le nom, la seconde vers `stdout`. La seule différence entre ces deux instructions est le premier argument. Une fois que vous aurez fait tourner le programme, utilisez un éditeur de texte pour voir ce que contient le fichier cocorico.txt (ou tout autre nom que vous aurez choisi). Ce fichier devrait se trouver dans le même répertoire que les fichiers du programme. Vous pourrez remarquer qu'il contient exactement ce qui a été affiché sur l'écran.

Vous aurez noté la présence de la fonction `clear kb()` que nous avons étudiée au Chapitre 14. Elle sert à supprimer d'éventuels caractères subsistant après l'appel à `scanf()` , qui viendraient perturber la lecture du nom de fichier. Il en résulterait une erreur au moment de son ouverture.

Entrées formatées à partir d'un fichier

Pour traiter une entrée d'informations à partir d'un fichier sur disque, on appelle la fonction de bibliothèque `fscanf()` qui s'utilise exactement comme `scanf()`, que nous avons étudiée au Chapitre 14, à un détail près : les informations proviennent cette fois, non plus du clavier (`stdin`), mais d'un fichier sur disque préalablement ouvert. Le prototype de `fscanf()` se trouve dans le fichier d'en-tête stdio.h et se présente ainsi :

```
int fscanf(FILE *fp, char *fmt,...);
```

Le premier argument est un pointeur vers un objet de type `FILE`. Pour lire des informations à partir d'un fichier sur disque particulier, il faut passer à la fonction le pointeur récupéré au moment de l'ouverture de ce fichier par `fopen()`.

Le deuxième argument est une chaîne de caractères contenant le format à utiliser. Vous avez déjà rencontré ce type de chaîne de caractères, lorsque nous avons étudié `scanf()`, au Chapitre 14. Nous retrouvons ici exactement le même type de chaîne.

Le dernier argument, `...`, signifie qu'on trouve ensuite un nombre variable d'arguments. Ces derniers sont les *adresses* des variables dans lesquelles `fscanf()` placera les valeurs lues sur le fichier disque.

Pour plus de détail, vous pouvez donc vous reporter au Chapitre 14.

Pour essayer `fscanf()`, vous devez disposer d'un fichier texte contenant quelques nombres ou chaînes de caractères dans un format acceptable par la fonction. À l'aide d'un

éditeur de texte, créez un fichier que vous appellerez, par exemple, infos.txt, dans lequel vous placerez cinq valeurs numériques exprimées en flottant et séparées par des espaces ou des retours à la ligne. Comme ceci :

```
123.45    87.001
100.02
0.000456  1.0005
```

Maintenant, compilez et lancez le programme du Listing 16.3.

Listing 16.3 : Utilisation de fscanf() pour lire des données formatées à partir d'un fichier sur disque

```
 1:  /* Lecture de données formatées sur un fichier avec fscanf(). */
 2:  #include <stdlib.h>
 3:  #include <stdio.h>
 4:
 5:  int main()
 6:  {
 7:      float f1, f2, f3, f4, f5;
 8:      FILE *fp;
 9:
10:      if ((fp = fopen("infos.txt", "r")) == NULL)
11:      {
12:          fprintf(stderr, "Erreur à l'ouverture du fichier.\n");
13:          exit(EXIT_FAILURE);
14:      }
15:
16:      fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
17:      printf("Les valeurs sont : %f, %f, %f, %f, et %f.\n",
18:              f1, f2, f3, f4, f5);
19:
20:      fclose(fp);
21:      exit(EXIT_SUCCESS);
22:  }
```



```
Les valeurs sont : 123.449997, 87.000999, 100.019997, 0.000456,
et 1.000500.
```

Rappelons que, par défaut, les valeurs affichées avec une spécification `%f` ont six chiffres après le point décimal (*K & R*, 2e édition, p. 154). Il en résulte la mise en évidence, comme dans le programme précédent, d'approximations de conversion.

Analyse

Le programme lit les cinq valeurs du fichier que vous avez créé et les affiche. À la ligne 10, `fopen()` ouvre le fichier en mode lecture et teste le résultat. En cas d'erreur, un message d'erreur est affiché et le programme se termine (lignes 12 et 13). La ligne 16 illustre

l'utilisation de la fonction `fscanf()`. À l'exception du premier argument, elle est identique à la fonction `scanf()` que nous avons déjà utilisée.

Entrées et sorties par caractères

Pour des fichiers sur disque, cela s'applique aussi bien à des caractères isolés qu'à des lignes de caractères. Souvenez-vous qu'une ligne est une suite d'un nombre variable de caractères (éventuellement zéro) terminée par le caractère `\n`. Ce mode s'utilise avec des fichiers texte.

Entrées par caractères

Il existe trois fonctions d'entrée de caractères : `getc()` et `fgetc()` pour les caractères isolés et `fgets()` pour les suites de caractères.

Les fonctions getc() et fgetc()

Les fonctions `getc()` et `fgetc()` sont identiques et peuvent être utilisées l'une à la place de l'autre. Elles acceptent un caractère isolé prélevé dans le flot spécifié. Le prototype de `getc()` se trouve dans stdio.h :

```
int getc(FILE *fp);
```

L'argument `fp` est le pointeur renvoyé par `fopen()` lors de l'ouverture du fichier. La fonction renvoie le caractère lu, ou `EOF` en cas d'erreur.

`getc()` a été utilisé dans des programmes précédents pour lire un caractère à partir du clavier. Nous avons ici un autre exemple de la souplesse des flots de C : la même fonction est utilisée pour lire à partir du clavier ou d'un fichier.

Si les fonctions `getc()` et `fgetc()` ne renvoient qu'un caractère, pourquoi leurs prototypes mentionnent-ils le retour d'un type `int` ? Lorsque vous lisez des fichiers, vous devez être capable de détecter le caractère de fin de fichier qui est déclaré en type `int` plutôt que `char` sur certains systèmes. Le programme du Listing 16.10 fait appel à la fonction `getc()`.

La fonction fgets()

Cette fonction de bibliothèque permet de lire une ligne de caractères à partir d'un fichier sur disque. Son prototype se trouve dans stdio.h :

```
char *fgets(char *str, int n, FILE *fp);
```

L'argument `str` est un pointeur vers une mémoire tampon dans lequel sera placée la ligne de caractères lue ; `n` est le nombre maximum de caractères à lire et `fp` est le pointeur de type `FILE` renvoyé par `fopen()` lors de l'ouverture du fichier.

`fgets()` lit des caractères dans le flot `fp` et les range en mémoire, depuis l'adresse pointée par `str` jusqu'à la rencontre d'un `\n` ou jusqu'à concurrence de `n   1` caractères. En fixant la longueur maximale de la chaîne à lire, on évite tout risque d'écrasement intempestif du contenu de la mémoire. C'est d'ailleurs pour cette raison que, pour le flot `stdin`, vous devez toujours utiliser `fgets()` et jamais `gets()`. Il reste un octet pour placer le `\0` terminal que `fgets()` insère systématiquement en fin de chaîne. En cas de succès, `fgets()` renvoie `str`. Deux types d'erreurs peuvent se produire :

S'il survient une erreur ou une fin de fichier sans qu'aucun caractère ne soit rangé dans `str`, `fgets()` renvoie `NULL`, et la zone de mémoire pointée par `str` reste inchangée.

S'il survient une erreur ou une fin de fichier après qu'un ou plusieurs caractères aient été rangés dans str, `fgets()` renvoie `NULL` et la zone de mémoire pointée par str contient n'importe quoi.

Vous voyez que `fgets()` ne lit pas nécessairement la totalité d'une ligne (c'est-à-dire jusqu'au `\n` final). Il lui suffit d'avoir lu `n   1` caractères pour se terminer normalement. L'opération suivante portant sur ce même fichier se poursuivra à l'endroit précis où la précédente s'est arrêtée. Pour être sûr d'avoir lu une ligne entière avec `fgets()`, il faut dimensionner le buffer de lecture à une valeur suffisante, en accord avec la valeur donnée à `n`.

Sortie de caractères

Nous allons voir deux fonctions de sortie de caractères : `putc()` et `fputs()`.

La fonction putc()

La fonction `putc()` écrit un unique caractère dans le flot spécifié. Son prototype se trouve dans stdio.h :

```
int putc(int ch, FILE *fp);
```

L'argument `ch` représente le caractère à écrire. Comme dans les autres fonctions appliquées à des caractères, on l'appelle de façon formelle un `int`, mais seul l'octet de poids faible est utilisé. L'argument `fp` est le pointeur de type `FILE` renvoyé lors de l'ouverture du fichier par `fopen()`. La fonction `putc()` renvoie le caractère qu'elle vient d'écrire en cas de réussite, ou `EOF` dans le cas contraire. La constante symbolique EOF est définie dans stdio.h comme ayant la valeur –1. Aucun véritable caractère n'ayant cette valeur, il n'y a pas de risque de confusion.

La fonction fputs()

Pour écrire une ligne de caractères dans un flot, on utilise la fonction de bibliothèque `fputs()`. Cette fonction est semblable à `puts()` que nous avons vue au Chapitre 14. La seule différence est que `fputs()` permet de spécifier un flot de sortie alors que `puts()` travaille toujours avec `stdout`. La fonction `fputs()` ne rajoute pas de caractère `\n` à la fin de la chaîne. Son prototype se trouve dans stdio.h :

```
char fputs(char *str, FILE *fp);
```

L'argument `str` est un pointeur vers la chaîne de caractères à écrire qui doit être terminée par un zéro, et `fp` est le pointeur de type `FILE` renvoyé par `fopen()` lors de l'ouverture du fichier. En cas de réussite, `fputs()` renvoie une valeur non négative. En cas d'erreur, elle renvoie `EOF`.

Entrées sorties directes

Cette méthode d'entrées sorties ne concerne que les fichiers en mode binaire. En sortie, la mémoire est écrite telle quelle dans les blocs d'informations sur disque. En entrée, la mémoire est garnie avec le contenu des blocs sur disque sans aucune conversion. Un seul appel de la fonction de sortie peut écrire un tableau entier de valeurs de type `double` sur le disque, et il suffira d'un seul appel de la fonction correspondante de lecture pour le réinstaller plus tard en mémoire. Les deux fonctions d'entrées-sorties directes sont `fread()` et `fwrite()`.

La fonction *fwrite()*

La fonction de bibliothèque `fwrite()` écrit un bloc d'informations à partir de la mémoire vers un fichier ouvert en mode binaire. Son prototype se trouve dans stdio.h :

```
int fwrite(void *buf, int size, int count, FILE *fp);
```

L'argument `buf` est un pointeur vers la région de la mémoire contenant les informations à écrire dans le fichier. Ce pointeur est de type `void`, c'est-à-dire qu'il peut pointer vers n'importe quoi.

L'argument `size` spécifie la taille (en nombre d'octets) des éléments à écrire, et `count`, leur nombre. Ainsi, si vous voulez sauvegarder un tableau de 100 valeurs numériques de type entier, `size` vaudra 4 (taille d'un `int`) et `count` vaudra 100. Vous pouvez utiliser l'opérateur `sizeof()` pour connaître la valeur de `size`.

L'argument `fp` est le pointeur de type `FILE` renvoyé par `fopen()` lors de l'ouverture du fichier. En cas de réussite, `fwrite()` renvoie le nombre d'articles écrits. En cas

d’erreur, elle renvoie une valeur inférieure. Pour voir si tout s’est bien passé, on peut écrire :

```
if(fwrite(buf, size, count, fp) != count)
  fprintf(stderr, "Erreur d'écriture sur disque.");
```

Voici quelques exemples d’utilisation de `fwrite()`. Pour écrire une seule variable `x` de type `double` dans un fichier, on écrira :

```
fwrite(&x, sizeof(x), 1, fp);
```

Pour écrire un tableau `data[]` de 50 structures de type `address` sur disque, deux formes sont possibles :

```
fwrite(data, sizeof(address), 50, fp);
fwrite(data, sizeof(data), 1, fp);
```

Dans le premier cas, on écrit le tableau sous forme de 50 éléments ayant chacun la taille d’une structure `address`. Dans le second cas, on traite le tableau comme un élément unique. Le résultat est le même dans les deux cas.

La fonction *fread()*

La fonction de bibliothèque `fread()` lit un bloc d’informations, à partir d’un fichier ouvert en mode binaire, dans une zone de mémoire. Son prototype se trouve dans stdio.h :

```
int fread(void *buf, int size, int count, FILE *fp);
```

L’argument `buf` est un pointeur vers la région de mémoire qui recevra les informations lues dans le fichier. Comme pour `fwrite()`, il est de type `void`.

L’argument `size` spécifie la taille (en nombre d’octets) des éléments à lire et `count`, leur nombre, comme pour `fwrite()`. On peut utiliser l’opérateur `sizeof()` pour connaître la valeur de `size`.

L’argument `fp` est le pointeur de type `FILE` renvoyé par `fopen()` lors de l’ouverture du fichier. En cas de réussite, `fread()` renvoie le nombre d’articles écrits. En cas d’erreur (par exemple, s’il ne reste plus assez de données à lire), elle renvoie une valeur inférieure.

Le programme du Listing 16.4 montre un exemple d’utilisation de `fwrite()` et de `fread()`.

Listing 16.4 : Utilisation de fwrite() et de fread() pour réaliser des accès directs sur disque

```
/* Entrées-sorties directes avec fwrite() et fread(). */
#include <stdio.h>
#include <stdlib.h>

#define SIZE 20

int main()
{
    int count, array1[SIZE], array2[SIZE];
     FILE *fp;

     /* Initialiser array1[]. */

     for (count = 0; count < SIZE; count++)
         array1[count] = 2 * count;

     /* Ouvrir un fichier en mode binaire. */

     if ((fp = fopen("direct.txt", "wb")) == NULL)
     {
         fprintf(stderr, "Erreur à l'ouverture du fichier.");
         exit(EXIT_FAILURE);
     }
     /* Sauvegarder array1[] dans le fichier. */

     if (fwrite(array1, sizeof(*array1), SIZE, fp) != SIZE)
     {
         fprintf(stderr, "Erreur à l'écriture du fichier.");
         exit(EXIT_FAILURE);
     }

     fclose(fp);

     /* Ouvrir maintenant le même fichier en mode binaire . */

     if ((fp = fopen("direct.txt", "rb")) == NULL)
     {
         fprintf(stderr, "Erreur à l'ouverture du fichier.");
         exit(EXIT_FAILURE);
     }

     /* Lire les informations dans array2[]. */

     if (fread(array2, sizeof(*array2), SIZE, fp) != SIZE)
     {
         fprintf(stderr, "Erreur … la lecture du fichier.");
         exit(EXIT_FAILURE);
     }
```

```
49:
50:       fclose(fp);
51:
52:       /* Afficher maintenant les deux tableaux pour montrer
53:          que ce sont les mêmes. */
54:       for (count = 0; count < SIZE; count++)
55:           printf("%d\t%d\n", array1[count], array2[count]);
56:       exit(EXIT_SUCCESS);
57:  }
```

```
0       0
2       2
4       4
6       6
8       8
10      10
12      12
14      14
16      16
18      18
20      20
22      22
24      24
26      26
28      28
30      30
32      32
34      34
36      36
38      38
```

Analyse

Après avoir initialisé un tableau aux lignes 14 et 15, le programme le sauvegarde sur disque à la ligne 26 avec une instruction `fwrite()`. Puis il relit le contenu du fichier dans un autre tableau à la ligne 44 avec `fread()`. Les deux tableaux sont affichés aux lignes 54 et 55.

Avec `fwrite()`, il ne peut pas arriver grand-chose en dehors d'une erreur d'écriture sur disque. En revanche, avec `fread()`, il faut bien savoir ce qu'on fait. La fonction n'a, en effet, aucun moyen de savoir quel type d'informations elle lit. Par exemple, un bloc de 100 octets pourrait être constitué de 100 caractères, ou de 50 entiers, ou encore de 25 flottants. Il faut donc faire la lecture dans un tableau de même type que les données contenues dans le fichier sur disque. Si on se trompe, on n'obtiendra aucune indication d'erreur, mais les données ainsi lues seront aberrantes. Dans cet exemple, vous noterez que chaque appel a une fonction d'entrées-sorties [`fopen()`, `fwrite()`, `fread()`] est suivi d'un test d'erreur.

Entrées-sorties tamponnées

Lorsqu'on a fini d'utiliser un fichier, il faut le refermer en appelant `fclose()`. Nous avons déjà rencontré cette fonction dont le prototype se trouve dans stdio.h :

```
int fclose(FILE *fp);
```

Son argument `fp` est toujours le pointeur de type `FILE` renvoyé lors de l'ouverture initiale du fichier. La fonction renvoie 0 si tout s'est bien passé, ou EOF en cas d'erreur. Lors de cet appel, les tampons sont purgés, c'est-à-dire que leur contenu est écrit sur disque. Il est possible de refermer tous les fichiers ouverts (à l'exclusion de `stdin`, `stdout` et `stderror`) par un seul appel à `fcloseall()`. Son prototype se trouve dans stdio.h :

```
int fcloseall(void)
```

La fonction renvoie le nombre de fichiers qu'elle a fermés.



`fcloseall()` est une extension GNU qui n'a rien de standard !

Lorsqu'un programme se termine, soit en atteignant la fin du `main()`, soit par un appel à `exit()`, tous les flots d'entrées-sorties sont automatiquement fermés. Cependant, il est préférable de les refermer explicitement par un appel à l'une des deux fonctions que nous venons de voir à cause de l'utilisation de tampons pour tamponner les entrées-sorties de flots.

Lorsqu'on crée un flot lié à un fichier sur disque, il y a automatiquement scréation d'une mémoire tampon associée à ce flot. Un tampon est un bloc de mémoire utilisé comme moyen de stockage temporaire pour les opérations d'entrées-sorties associées au flot.

Les tampons sont nécessaires parce que les disques sont des périphériques de type bloc ; cela signifie qu'on ne lit pas sur disque n'importe quel nombre d'octets, mais un nombre constant correspondant à la structure logique adoptée sur le disque (celle-ci dépend elle-même du système d'exploitation).

Le tampon associé au fichier sert d'interface entre le disque (de type "bloc") et le flot (de type "à caractère"). Lors d'une écriture, les informations sont accumulées dans la mémoire tampon et ne seront écrites que lorsque celle-ci sera pleine. Le même processus est utilisé, dans l'autre sens, pour la lecture. Le langage C dispose de certaines fonctions permettant d'agir sur la gestion des tampons, mais leur étude sortirait du cadre de ce livre.

Il résulte de ces considérations qu'on ne sait jamais à quel moment auront lieu les véritables opérations de lecture ou d'écriture sur disque. Si le programme se bloque, ou en cas de coupure de courant, on risque de perdre des informations.

Il est possible de purger (*flushing*) les tampons d'un flot sans refermer celui-ci par un appel aux fonctions de bibliothèque `fflush()`. Le prototype de cette fonction se trouve dans stdio.h :

```
int fflush(FILE *fp);
```

L'argument `fp` est le pointeur renvoyé par `fopen()` lors de l'ouverture du fichier. Si celui-ci était ouvert en écriture, le contenu du tampon est écrit sur disque. S'il était ouvert en lecture, le contenu du tampon est simplement purgé. La fonction `fflush()` renvoie 0 si tout s'est bien passé, ou EOF en cas d'erreur.



À faire

Ouvrir un fichier avant d'essayer de le lire ou d'y écrire.

Utiliser `sizeof()` dans les appels de `fread()` et `fwrite()`.

Refermer explicitement les fichiers qui ont été ouverts.

Ne pas faire

Supposer implicitement correcte une opération sur un fichier. Tester toujours le résultat.

Accès séquentiel opposé à accès direct

Un indicateur de position est associé à chaque fichier ouvert. Il indique à quel endroit du fichier aura lieu les prochaines opérations de lecture et d'écriture. La position est donnée en nombre d'octets à partir du début du fichier. Lorsqu'on crée un nouveau fichier, l'indicateur de position vaut 0 puisque le fichier ne contient encore rien. Lorsqu'on ouvre un fichier qui existe déjà, l'indicateur de position indique la fin du fichier si celui-ci est ouvert en mode "mise à jour". Dans tout autre mode, il indique le début du fichier.

Les fonctions que nous venons d'étudier exploitent et mettent à jour cet indicateur. Lecture et écriture s'effectuent à l'endroit correspondant à sa valeur. Ainsi, si vous ouvrez un fichier en lecture et que vous lisez 10 octets, l'indicateur de position prend la valeur 10 et c'est là qu'aura lieu la lecture suivante. Pour lire séquentiellement tout le fichier, vous n'avez pas à vous préoccuper de son indicateur de position.

Lorsque vous souhaitez exercer un contrôle plus précis sur la suite des opérations, utilisez les fonctions de la bibliothèque standard du C qui permettent de manipuler cet indicateur. Vous pouvez, de cette façon, effectuer un *accès aléatoire* à votre fichier, autrement dit, lire ou écrire des blocs par-ci, par-là, sans que leurs positions soient nécessairement consécutives.

Les fonctions *ftell()* et *rewind()*

Pour manipuler l'indicateur de position associé à un fichier, on utilise les fonctions de bibliothèque `ftell()` et `rewind()` dont les prototypes se trouvent dans stdio.h :

```
void rewind(FILE *fp);
```

qui place l'indicateur de position au début du fichier et

```
long ftell(FILE *fp);
```

qui permet de connaître la valeur de l'indicateur de position.

Dans ces deux fonctions, l'argument `fp` est le pointeur renvoyé par `fopen()` lors de l'ouverture du fichier. L'entier de type `long` renvoyé par `ftell()` indique le nombre d'octets séparant la position actuelle du début du fichier. En cas d'erreur, la fonction renvoie –`1L`.

Le programme du Listing 16.5 vous donne un exemple d'utilisation de ces deux fonctions.

Listing 16.5 : Utilisation de ftell() et rewind()

```
/* Démonstration de ftell() et rewind(). */
#include <stdio.h>
#include <stdlib.h>

#define BUFLEN 6

char msg[] = "abcdefghijklmnopqrstuvwxyz";

int main()
{
    FILE *fp;
    char buf[BUFLEN];

    if ((fp = fopen("texte.txt", "w")) == NULL)
    {
          fprintf(stderr, "Erreur à l'ouverture du fichier.");
          exit(EXIT_FAILURE);
    }

    if (fputs(msg, fp) == EOF)
    {
        fprintf(stderr, "Erreur à l'écriture du fichier.");
        exit(EXIT_FAILURE);
    }

    fclose(fp);

    /* Ouvrons maintenant le fichier en lecture. */

    if ((fp = fopen("texte.txt", "r")) == NULL)
    {
```

```
32:             fprintf(stderr, "Erreur à l'ouverture du fichier.");
33:             exit(EXIT_FAILURE);
34:       }
35:       printf("\nImmédiatement après l'ouverture, position = %ld",
36:               ftell(fp));
37:       /* Lire 5 caractères. */
38:
39:       fgets(buf, sizeof(buf), fp);
40:       printf("\nAprès lecture de %s, position = %ld", buf,
41:               ftell(fp));
42:       /* lire les 5 caractères suivants. */
43:
44:       fgets(buf, sizeof(buf), fp);
45:       printf("\n\nLes 5 caractères suivant sont %s. \
46:               Position maintenant = %ld", buf, ftell(fp));
47:
48:       /* "Rembobiner" le flot. */
49:
50:       rewind(fp);
51:
52:       printf("\n\nAprès rembobinage, la position est revenue \
53:               à %Ld", ftell(fp));
54:
55:       /* Lire 5 caractères. */
56:
57:       fgets(buf, sizeoff(buf), fp);
58:       printf("\net la lecture commence au début à nouveau : %s\n", buf);
59:       fclose(fp);
60:       exit(EXIT_SUCCESS);
61:   }
```



```
Immédiatement après l'ouverture, position = 0
Après lecture de abcde, position = 5

Les 5 caractères suivants sont fghij. Position maintenant = 10

Après rembobinage, la position est revenue à 0
et la lecture commence au début à nouveau : abcde
```

Analyse

Ce programme écrit la chaîne de caractères `msg` (les 26 lettres de l'alphabet dans l'ordre) dans un fichier appelé texte.txt qui va être ouvert aux lignes 14 à 18 en sortie. On s'assure, bien entendu, que la création s'est bien passée. Les lignes 20 à 24 écrivent `msg` dans le fichier à l'aide de la fonction `fputs()`, et vérifient la réussite de l'opération. Le fichier est refermé à la ligne 26, ce qui termine le processus de création.

Le fichier est ensuite ouvert en lecture (lignes 30 à 34). La valeur retournée par `ftell()` est affichée à la ligne 35. La ligne 39 effectue une lecture de cinq caractères par `fgets()`. Ces cinq caractères ainsi que la nouvelle position du fichier sont affichés à la ligne 40. La fonction `rewind()` est appelée à la ligne 50 pour replacer le fichier à son début. On affiche

à nouveau sa position à la ligne 52. Une nouvelle lecture (ligne 57) confirme la valeur à laquelle on s'attendait. Le fichier est refermé à la ligne 59, avant la fin du programme.

La fonction *fseek()*

On peut exercer un contrôle plus précis sur l'indicateur de position d'un flot en appelant la fonction de bibliothèque `fseek()` qui permet de lui donner une nouvelle valeur. Le prototype de cette fonction se trouve dans stdio.h :

```
int fseek(FILE *fp, long offset, int origin);
```

(voir Tableau 16.2).

Tableau 16.2 : Valeurs d'origine possible pour fseek().

Constante	*Valeur*	*Signification*
`SEEK SET`	`0`	Début du fichier
`SEEK CUR`	`1`	Position courante
`SEEK END`	`2`	Fin du fichier

L'appel à `fseek()` renvoie 0 si l'indicateur a réellement pris la valeur demandée, ou une valeur non nulle dans le cas contraire. On voit, sur le Listing 16.6, comment utiliser la fonction `fseek()`.

Listing 16.6 : Accès aléatoire à un fichier à l'aide de la fonction fseek()

```
/* Accès aléatoire avec fseek(). */

#include <stdio.h>
#include <stdlib.h>

#define MAX 50

int main()
{
    FILE *fp;
    int data, count, array[MAX];
    long offset;

    /* Initialiser le tableau. */

    for (count = 0; count < MAX; count++)
        array[count] = count * 10;

    /* Ouvrir un fichier binaire en écriture. */
```

```c

    if ((fp = fopen("random.dat", "wb")) == NULL)
    {
        fprintf(stderr, "\nErreur à l'ouverture du fichier.");
        exit(EXIT_FAILURE);
    }

    /* Écrire le tableau dans le fichier puis le refermer . */

    if ((fwrite(array, sizeof(*array), MAX, fp)) != MAX)
    {
        fprintf(stderr, "\nErreur à l'écriture dans le fichier.");
        exit(EXIT_FAILURE);
    }

    fclose(fp);

    /* Ouvrir le fichier en lecture. */

    if ((fp = fopen("random.dat", "rb")) == NULL)
    {
        fprintf(stderr, "\nErreur à l'ouverture du fichier.");
        exit(EXIT_FAILURE);
    }

    /* Demander à l'utilisateur quel élément il veut lire. Lire
       l'élément et l'afficher. Arrêter lorsqu'il répond -1. */

    while (1)
    {
        printf("\nIndiquez l'élément à lire, 0-%d, -1 pour \
                arrêter : ", MAX-1);
        scanf("%ld", &offset);
        if (offset < 0)
            break;
        else if (offset > MAX-1)
            continue;

        /* Déplacer l'indicateur de position sur l'élément
           spécifié. */
        if (fseek(fp, (offset*sizeof(int)), SEEK_SET))
        {
            fprintf(stderr, "\nErreur avec fseek().");
            exit(EXIT_FAILURE);
        }

        /* Lire un unique entier. */

        fread(&data, sizeof(data), 1, fp);

        printf("\nL'élément %ld a la valeur %d.", offset, data);
    }

    fclose(fp);
    exit(EXIT_SUCCESS);
}
```

```
Indiquez l'élément à lire, 0-49, -1 pour arrêter : 5

L'élément 5 a la valeur 50.
Indiquez l'élément à lire, 0-49, -1 pour arrêter : 6

L'élément 6 a la valeur 60.
Indiquez l'élément à lire, 0-49, -1 pour arrêter : 49

L'élément 49 a la valeur 490.
Indiquez l'élément à lire, 0-49, -1 pour arrêter : 10

L'élément 1 a la valeur 10.
Indiquez l'élément à lire, 0-49, -1 pour arrêter : 0

L'élément 0 a la valeur 0.
Indiquez l'élément à lire, 0-49, -1 pour arrêter : -1
```

Analyse

Le début du programme est identique à celui du programme précédent. Ici, le fichier porte le nom de random.dat. Une fois qu'on a terminé l'écriture, on le referme (ligne 35) pour le rouvrir en lecture (ligne 39). En cas (improbable) d'erreur, un message est affiché et le programme se termine immédiatement. Ensuite, dans une boucle `while` perpétuelle (lignes 48 à 71), on demande à l'utilisateur d'indiquer une valeur comprise entre 0 et 49 (lignes 50 et 52). Cette valeur est comparée aux limites des enregistrements écrits (0 à 49) par les instructions des lignes 53 à 56. Si elle est négative, `break` permet de sortir de la boucle `while`. Si elle est supérieure à `MAX 1`, on saute ce qui suit et on remonte en tête de la boucle `while` où on redemande une autre valeur.

Lorsque la valeur est reconnue bonne, on place le fichier à l'endroit demandé (ligne 60), on lit l'entier qui se trouve à cet endroit (ligne 68), puis on l'affiche (ligne 70). Ensuite, la remontée normale dans la boucle `while` s'effectue.

Détection de la fin d 'un fichier

Lorsque vous connaissez exactement la longueur du fichier que vous souhaitez lire, il n'est pas nécessaire de pouvoir en détecter la fin. Par exemple, si vous avez sauvegardé un tableau de cent entiers, vous savez que le fichier a une longueur de 200 octets. Mais, dans certains cas, vous ignorez la longueur exacte du fichier tout en voulant, quand même, le lire du début jusqu'à la fin. Il existe deux moyens de détecter une fin de fichier.

Lorsque vous lisez un fichier écrit en mode texte, caractère par caractère, vous pouvez tester son caractère de fin. La constante symbolique `EOF` est définie dans stdio.h comme

ayant une valeur égale à –1, valeur ne correspondant à aucun caractère réel. Dès lors, vous pouvez écrire :

```
while ((c = fget(fp)) != EOF)
{
...
}
```

En mode binaire, cette méthode n'est pas applicable puisqu'il n'y a pas de valeur "réservée" et qu'on peut lire indifféremment toute association de bits. Il existe heureusement la fonction de bibliothèque `feof()` (utilisable dans les deux modes, binaire et texte), qui est ainsi définie :

```
int feof(FILE *fp);
```

Elle renvoie 0 lorsqu'on ne se trouve pas à la fin du fichier, ou une valeur non nulle si on y est. À ce moment, aucune autre opération de lecture n'est possible, tout au moins, tant qu'un `rewind()` ou un `fseek()` n'a pas été effectué ou que le fichier n'a pas été fermé puis rouvert.

Le programme du Listing 16.7 montre comment utiliser `feof()`. Lorsque le programme vous demande un nom de fichier, donnez-lui le nom d'un fichier texte (un de vos programmes C, par exemple). Assurez-vous que ce fichier existe bien dans le répertoire courant ou précisez le chemin d'accès avec le nom. Il va lire le fichier d'un bout à l'autre, en l'affichant ligne par ligne, jusqu'à ce qu'il atteigne la fin du fichier.

Listing 16.7 : Utilisation de la fonction feof() pour détecter une fin de fichier

```
/* Détection d'une fin de fichier. */

#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE 100

int main()
{
    int k;
    char buf[BUFSIZE];
    char filename[60];
    FILE *fp;

    puts("Indiquez le nom du fichier texte à afficher : ");
    lire_clavier(filename, sizeof(filename));

    /* Ouverture du fichier en lecture. */
    if ((fp = fopen(filename, "r")) == NULL)
    {
        fprintf(stderr, "Erreur à l'ouverture du fichier.");
        exit(EXIT_FAILURE);
```

Listing 16.7 : Utilisation de la fonction feof() pour détecter une fin de fichier *(suite)*

```
23:        }
24:
25:        /* Lire une ligne et, si on n'est pas à la fin du fichier,
26:           l'afficher. */
27:
28:        do
29:        {  fgets(buf, sizeof(buf), fp);
30:           if (!(k = feof(fp))) printf("%s", buf);
31:        } while (k);
32:
33:        fclose(fp);
34:        exit(EXIT_SUCCESS);
35:    }
```

Voici un exemple d'exécution de ce programme :

```
Indiquez le nom du fichier texte à afficher :
hello.c
#include <stdio.h>
#include <stdlib.h>
main()
{
    printf("Bonjour le monde.");
    exit(EXIT_SUCCESS);
}
```

Analyse

On rencontre une boucle `do while` comme celle des lignes 28 à 31 dans les programmes effectuant un traitement séquentiel. Il faut tester la fin de fichier immédiatement après la lecture et avant d'avoir imprimé quoi que ce soit, car ce n'est qu'à ce moment-là qu'on saura si on a réellement lu quelque chose. Si on est parvenu à la fin du fichier, il faut donc éviter d'imprimer et, en fin de boucle, ne pas remonter. La variable `k`, dans laquelle est mémorisé le résultat du test, va donc gouverner les deux instructions : l'affichage et la remontée dans la boucle.

Il faut noter que cette solution suppose implicitement qu'il ne se produit pas d'erreur sur la lecture du fichier. Si c'était le cas, on ne sortirait pas de la boucle `do while`.

Pour tester le programme, on constituera un fichier qu'on appellera toto (par exemple, avec votre éditeur de texte) dans lequel on placera simplement ces trois lignes :

```
abcd
efgh
ijkl
```

puis on lancera le programme. On verra alors s'afficher :

```
Indiquez le nom du fichier texte à afficher : toto
abcd
efgh
ijkl
```



À faire

Utiliser `rewind()` ou `fseek()` pour replacer le fichier à son début.

Utiliser `feof()` pour tester la fin du fichier lorsqu'on travaille sur des fichiers binaires.

À ne pas faire

Utiliser `EOF` sur des fichiers binaires (sur Linux, les fichiers sont tous considérés comme binaires).

Fonctions de gestion de fichier

L'expression "gestion de fichiers" concerne d'autres opérations que la lecture et l'écriture : l'effacement, le changement de nom ou la recopie. La bibliothèque standard du C contient des fonctions permettant d'effacer un fichier ou d'en changer le nom, et rien ne vous empêche d'écrire vous-même des fonctions de recopie.

Effacement d'un fichier

Pour effacer un fichier, il faut utiliser la fonction de bibliothèque `remove()`. Elle est définie dans stdio.h et voici son prototype :

```
int remove (const char *filename);
```

La variable `filename` est un pointeur vers une chaîne de caractères contenant le nom du fichier à effacer. Ce fichier ne doit pas être ouvert. S'il existe, il est effacé, comme si vous aviez utilisé la commande DEL (Windows) ou `rm (Unix)`, d'où, d'ailleurs, cette instruction tire son nom), et la fonction renvoie 0. Si le fichier n'existe pas ou possède un attribut *read only* (lecture seulement), si vous ne possédez pas les droits d'accès requis, ou si une autre erreur survient, la fonction renvoie –1.

Le court programme du Listing 16.8 montre comment utiliser `remove()`. Évitez d'utiliser, pour le tester, un fichier auquel vous tenez !

Listing 16.8 : Utilisation de la fonction remove() pour effacer un fichier sur disque

```
 1:   /* Démonstration de la fonction remove(). */
 2:   #include <stdio.h>
 3:   #include <stdlib.h>
 4:
 5:   int main()
 6:   {
 7:       char filename[80];
 8:
 9:       printf("Indiquez le nom du fichier à supprimer : ");
10:       lire_clavier(filename, sizeof(filename));
11:
12:       if (remove(filename) == 0)
13:           printf("Le fichier %s a été supprimé.\n", filename);
14:       else
15:           fprintf(stderr, "Erreur à la suppression du fichier \
16:                   %s.\n", filename);
17:       exit(EXIT_SUCCESS);
18:   }
```



```
Indiquez le nom du fichier à supprimer : toto.bak
Le fichier toto.bak a été supprimé.
```

Analyse

Le programme demande à l'utilisateur d'indiquer le nom du fichier à effacer à la ligne 9. L'appel à `remove()` s'effectue à la ligne 12. Selon la valeur de retour, un message ou un autre est alors affiché.

La fonction `unlink()` existe également. C'est d'ailleurs à celle-ci que `remove()` fait appel.

Changement du nom d'un fichier

La fonction `rename()` permet de changer le nom d'un fichier. Son prototype est dans stdio.h :

```
int rename(const char *oldname, const char *newname);
```

Le fichier dont le nom est pointé par `oldname` prend le nom pointé par `newname`. Pour être certain d'obtenir un résultat correct avec tous les compilateurs, ces deux noms ne doivent comporter aucune indication d'unité de disque ou de chemin d'accès. Cette fonction admet qu'un chemin d'accès différent soit indiqué pour chacun des noms, ce qui réalise en même temps un déplacement du fichier. Le compilateur Microsoft Visual C++ admet même que l'unité de disque soit différente pour chacun des deux fichiers mais ce comportement n'est pas standard. La fonction renvoie 0 si tout s'est

bien passé, –1 dans le cas contraire. Le Listing 16.9 présente un exemple de programme C utilisant cette fonction.

Les causes d'erreur les plus fréquentes sont :

- Le fichier `oldname` n'existe pas.
- Il existe déjà un fichier ayant le nom "newname".
- Vous avez indiqué un nom de disque ou un chemin d'accès (dépend du compilateur utilisé).

Listing 16.9 : Utilisation de la fonction rename() pour changer le nom d'un fichier sur disque

```
 1:   /* Utilisation de rename() pour changer le nom d'un fichier. */
 2:   #include <stdio.h>
 3:   #include <stdlib.h>
 4:
 5:   int main()
 6:   {
 7:        char oldname[80], newname[80];
 8:
 9:        printf("Indiquez le nom actuel du fichier : ");
10:       lire_clavier(oldname, sizeof(oldname));
11:       printf("Indiquez le nouveau nom du fichier : ");
12:       lire_clavier(newname, sizeof(newname));
13:
14:       if (rename(oldname, newname) == 0)
15:          printf("%s s'appelle maintenant %s.\n", oldname, newname);
16:       else
17:          fprintf(stderr, "Erreur survenue en changeant le nom \
18:                  de %s.\n",   oldname);
19:       exit(EXIT_SUCCESS);
20:   }
```



```
Indiquez le nom actuel du fichier : toto.txt
Indiquez le nouveau nom du fichier : titi.txt
titi.txt s'appelle maintenant titi.txt
```

Analyse

Le Listing 16.9 illustre la puissance du langage C. Avec seulement 18 lignes de code, le programme remplace une commande du système, et ce, de façon plus conviviale. À la ligne 9, on demande à l'utilisateur de donner le nom du fichier dont il veut changer le nom. À la ligne 11, on lui demande le nouveau nom qu'il veut lui donner et, à la ligne 14, on lui dit ce qui s'est passé, selon la valeur renvoyée par `rename()`.

Copie d'un fichier

Il est souvent nécessaire de faire une copie d'un fichier : un double sous un nom différent (ou sous le même nom, mais dans un autre répertoire ou sur un autre support). En ligne de commande, on dispose de `COPY` (Windows) ou `cp` (Unix). En C, il n'existe pas de fonction de bibliothèque pour cela ; aussi faut-il écrire soi-même un programme à cette fin.

À priori, cela pourrait vous paraître compliqué, mais il n'en est rien. Voici la marche à suivre :

1. Ouvrir le fichier source en lecture et en mode binaire (ce qui permet de recopier n'importe quel type de fichier).
2. Ouvrir le fichier destinataire en écriture et en mode binaire.
3. Lire quelques caractères dans le fichier source. À l'ouverture, on est certain que le fichier est placé à son début, donc, inutile de faire appel à `fseek()`.
4. Si un appel à `feof()` indique qu'on a atteint la fin du fichier, fermer les deux fichiers et terminer le programme.
5. Sinon, écrire les caractères sur le fichier destinataire et reprendre à l'étape 3.

Le programme du Listing 16.10 contient une fonction, `copy file()`, à laquelle on passe les noms du fichier source et du fichier destinataire, et qui effectue l'opération de copie selon le schéma que nous venons d'esquisser. Cette fonction n'est pas appelée en cas d'erreur à l'ouverture de l'un des deux fichiers. Une fois la copie terminée, les deux fichiers sont fermés et la fonction renvoie 0.

Listing 16.10 : Fonction recopiant un fichier

```
/* Copie d'un fichier. */
#include <stdio.h>
#include <stdlib.h>

int file_copy(char *oldname, char *newname);

int main()
{
     char source[80], destination[80];

    /* Demander les noms des fichiers source et destination. */

    printf("\nIndiquer le nom du fichier source : ");
    lire_clavier(source, sizeof(source));
    printf("\nIndiquez le nom du fichier destination : ");
    lire_clavier(destination, sizeof(destination));

    if (file_copy(source, destination) == 0)
        puts("Copie réussie");
```

```
      else
          fprintf(stderr, "Erreur au cours de la copie");
      exit(EXIT_SUCCESS);
}
  int file_copy(char *oldname, char *newname)
  {
      FILE *fold, *fnew;
      char buf[BUFSIZ];
      int n;
      /* Ouverture du fichier source en lecture, mode binaire. */
      if ((fold = fopen(oldname, "rb")) == NULL)
          return -1;
      /* Ouverture du fichier destination en écriture,
         en mode binaire. */
      if ((fnew = fopen(newname, "wb")) == NULL  )
      {
          fclose (fold);
          return -1;
      }
      /* Lire le fichier source morceaux par morceaux. Si on n'a pas
         atteint la fin du fichier, écrire les données sur le
         fichier destination. */
      while (!feof(fold))
      {
          n = fread(buf, 1, sizeof(buf), fold);
          if (n > 0)
              fwrite(buf, 1, n, fnew);
          else
              break;
      }
      fclose (fnew);
      fclose (fold);
      return 0;
  }
```

```
Indiquer le nom du fichier source : liste.doc

Indiquer le nom du fichier destination : sauvegarde.txt
Copie réussie
```

Analyse

La fonction `copy file()` permet de copier n'importe quoi, depuis un petit fichier texte jusqu'à un énorme fichier de programme. Elle a, cependant, quelques limites. Si le fichier

de destination existe déjà, la fonction l'efface sans demander la permission. Vous pourriez, à titre d'exercice, la modifier pour tester l'existence du fichier de destination et, dans ce cas, demander la permission de l'écraser.

Ici, `main()` est quasi identique au `main()` du Listing 16.9, à l'exception de la ligne 14. Ce n'est pas `rename()` qu'on appelle, mais `copy file()`. Les instructions de cette fonction se trouvent aux lignes 24 à 60. L'ouverture du fichier source se fait, en mode binaire, aux lignes 31 et 32 et celle du fichier destinataire aux lignes 36 à 40. Si une erreur se produit à ce moment, on referme le fichier source. La boucle `while` des lignes 46 à 54 effectue la recopie du fichier. La ligne 48 lit un bloc de caractères dans le fichier source, `fold`. À la ligne 50, on regarde si on a lu des données. Si oui, les données lues sont écrites sur le fichier de sortie, `fnew`. Sinon, on exécute un `break` afin de sortir de la boucle. On détecte la fin de fichier avec `feof()` en tant que condition de `while()` ligne 46. Aux lignes 56 et 57, on trouve deux instructions `fclose()` qui referment les fichiers.

Remarque : On aurait pu effectuer la copie octet par octet. Cela est à éviter pour des raisons évidentes de performances. Par ailleurs, la variable BUFSIZ est une variable définie dans `stdin.h`.

Emploi de fichiers temporaires

Certains programmes ont besoin d'un ou plusieurs fichiers de travail (temporaires) durant leur exécution. Un *fichier temporaire* est créé par le programme, utilisé à certaines fins pendant son exécution et supprimé juste avant que le programme se termine. Lorsque vous créez un fichier temporaire, son nom importe peu puisqu'il ne sera pas conservé. L'essentiel est de ne pas utiliser un nom de fichier existant déjà. Pour cela, il existe dans la bibliothèque standard C une fonction `mkstemp()` qui fabrique un nom de fichier réputé unique. Son prototype se trouve dans stdio.h :

```
int mkstemp(char *template);
```

Son argument est un pointeur vers un buffer contenant un modèle de fichier temporaire se terminant par "XXXXXX" (six fois la lettre X). Ce buffer est impérativement une chaîne de caractères créée avec la fonction `malloc()` (ou équivalent comme `strdup()` que nous allons utiliser ci-dessous). Vous penserez donc à libérer l'espace mémoire ainsi réservé. La fonction `mkstemp()` a pour rôle de créer un fichier temporaire et de l'ouvrir. Le nom du fichier est écrit en écrasant les "XXXXXX" par des caractères de façon à ce qu'il soit unique. Par ailleurs, `mkstemp()` renvoie un descripteur de fichier (ou –1 en cas d'erreur) que vous pouvez transformer en descripteur de flux avec la fonction `fdopen()`.

Le programme du Listing 16.11 montre comment utiliser cette méthode pour créer des noms de fichier temporaires.

Listing 16.11 : Utilisation de la fonction mkstemp() pour créer des noms de fichier temporaires

```
/* Démonstration de noms de fichier temporaires. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *buffer;
    int fd;
    FILE *tmpfd

    /* Garnir le buffer avec un nom de fichier temporaire. */

    buffer = strdup("fichier_XXXXXX");

    /* Créer le fichier temporaire */

    if((fd = mkstemp(buffer)) == -1)
    {
        fprintf(stderr, "Impossible de créer le fichier\n");
        exit(EXIT_FAILURE);
    }
    if((tmpfd = fdopen(fd, "wb")) == NULL)
    {
        fprintf(stderr, "Impossible de créer le flux\n");
        exit(EXIT_FAILURE);
    }

    /* Utiliser le fichier temporaire */
    /* ... */
    /* Afficher les noms. */

    printf("Nom de fichier temporaire : %s\n", buffer);

    /* Fermer le fichier et faire le ménage */

    fclose(tmpfd);
    free(buffer);

    exit(EXIT_SUCCESS);
}
```



```
Nom de fichier temporaire : fichier_njcU7l
```

Analyse

Les noms générés sur votre système peuvent être quelque peu différents de ceux que vous voyez ici.

Remarque : il existe plusieurs fonctions pour créer des fichiers temporaires ou des noms de fichiers temporaires. Leur principe d'utilisation les rend peu sûres. Utilisez `mkstemp()` ou mieux, `tmpfile()`. Nous n'avons pas présenté cette dernière car elle ne permet pas de connaître le nom du fichier temporaire ainsi créé. Si vous n'avez pas besoin de connaître le nom du fichier temporaire que vous voulez utiliser, préférez cette dernière dont le prototype est des plus simples :

```
FILE *tmpfile(void);
```



À ne pas faire

Supprimer un fichier qui pourrait être nécessaire plus tard.

Essayer de changer le nom d'un fichier sur un autre disque.

Oublier de supprimer les fichiers temporaires créés.

Résumé

Dans ce chapitre, vous avez appris à utiliser des fichiers sur disque dans un programme C. Dans ce langage, les fichiers sont considérés comme des flots, c'est-à-dire comme une séquence de caractères, de la même façon que les flots prédéterminés que vous avez découverts au Chapitre 14. On doit commencer par ouvrir un flot associé à un fichier sur disque avant de pouvoir l'utiliser, et il faut le refermer après usage. Un flot sur disque peut être ouvert en entrée ou en sortie.

Une fois le fichier sur disque ouvert, vous pouvez lire les informations qu'il contient ou y écrire d'autres informations, ou les deux. Il existe trois types d'entrées-sorties : formatées, à caractères et directes. Le choix entre ces trois types dépend de l'utilisation qu'on veut faire du fichier.

À chaque fichier sur disque se trouve associé un indicateur de position, qui indique le nombre d'octets séparant la position actuelle du début du fichier. Il précise l'endroit du fichier où aura lieu la prochaine opération d'entrées-sorties. Certaines de ces opérations mettent à jour l'indicateur automatiquement. Pour les fichiers en accès direct, la bibliothèque standard du C propose des fonctions permettant de les manipuler.

Il existe aussi des fonctions rudimentaires de gestion de fichier qui permettent de supprimer un fichier ou de changer son nom. Enfin, nous avons vu comment écrire un programme de recopie de fichier.

Q & R

Q **Puis-je spécifier un disque et un chemin d'accès avec le nom de fichier dans les opérations `remove()`, `rename()`, `fopen()` et les autres fonctions de traitement de fichier ?**

R Oui. Mais attention avec `rename()` : cette fonction sert également à déplacer un fichier d'un répertoire à d'autre d'un même disque. N'oubliez pas que l'anti-slash est un caractère d'échappement. Il faut donc le redoubler à l'intérieur d'une chaîne de caractères. Avec UNIX, le séparateur de répertoires est un slash ordinaire (`/`) et non un antislash (`\`).

Q **Peut-on lire des informations derrière une fin de fichier ?**

R Vous pouvez toujours essayer. Mais à vos risques et périls !

Q **Qu'arrive-t-il si je ne referme pas un fichier ?**

R Refermer un fichier après usage est une excellente habitude de programmation. En principe, le système d'exploitation referme les fichiers qui sont encore ouverts lorsqu'un programme se termine. Mais mieux vaut ne pas trop compter là-dessus. Si le fichier n'est pas refermé, avec certains systèmes d'exploitation (mais ni Unix, ni Linux ni Windows), vous pourriez ne plus pouvoir le rouvrir, car il serait considéré comme étant toujours en cours d'utilisation.

Q **Combien de fichiers puis-je ouvrir en même temps ?**

R C'est une question à laquelle on ne peut pas répondre simplement. Cela dépend essentiellement de certains paramètres du système d'exploitation.

Q **Puis-je lire un fichier séquentiel avec des fonctions prévues pour l'accès direct ?**

R Lorsqu'on lit un fichier en séquence, il n'est pas nécessaire d'utiliser des fonctions comme `fseek()`, car l'indicateur de position suit fidèlement le déroulement des opérations successives. Mais ce n'est pas défendu ; c'est seulement inutile.

Atelier

L'atelier vous propose quelques questions permettant de tester vos connaissances sur les sujets que nous venons d'aborder dans ce chapitre.

Quiz

1. Quelle est la différence entre un flot en mode texte et un flot en mode binaire ?
2. Que doit faire votre programme avant de pouvoir accéder à un fichier sur disque ?

3. Lorsque vous ouvrez un fichier avec `fopen()`, quelles informations devez-vous spécifier et que renvoie la fonction ?
4. Quelles sont les trois méthodes générales d'accès à un fichier ?
5. Quelles sont les deux méthodes générales pour lire les informations contenues dans un fichier ?
6. Que vaut `EOF` ?
7. À quel moment utilise-t-on `EOF` ?
8. Comment détecte-t-on la fin d'un fichier en mode texte et en mode binaire ?
9. Qu'est-ce que l'indicateur de position de fichier et comment peut-on modifier sa valeur ?
10. Lorsqu'on ouvre un fichier, où pointe l'indicateur de position ? (Si vous n'êtes pas sûr de votre réponse, voyez le Listing 16.5.)

Exercices

1. Indiquez deux façons de restaurer la valeur de l'indicateur de position au début d'un fichier.
2. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions qui suivent ?

```
FILE *fp;
int c;
if ((fp=fopen(oldname, "rb")) == NULL)
return -1;
while ((c = fgetc(fp)) != EOF)
fprintf(stdout, "%c", c);
fclose (fp);
```

Pour les exercices 4 à 8 qui suivent, il y a plusieurs solutions possibles. Nous n'en donnerons pas le corrigé.

4. Écrivez un programme qui affiche le contenu d'un fichier sur l'écran.
5. Écrivez un programme qui ouvre un fichier et compte le nombre de caractères qu'il contient. Une fois le fichier entièrement lu, ce nombre sera affiché. Contrôlez la solution avec l'utilitaire wc (`wc fichier`) sur Linux
6. Écrivez un programme qui ouvre un fichier texte existant et le recopie vers un nouveau fichier texte, en transformant toutes les lettres minuscules en majuscules et en laissant les autres caractères inchangés.

7. Écrivez un programme qui ouvre n'importe quel fichier sur disque, le lit par blocs de 128 octets et affiche le contenu de chaque bloc sur l'écran, à la fois en hexadécimal et sous forme de caractères ASCII.
8. Écrivez une fonction qui ouvre un fichier temporaire dans un mode spécifié. Tous les fichiers temporaires créés par cette fonction devront être automatiquement refermés et supprimés avant que le programme ne se termine. (Astuce : utilisez la fonction de bibliothèque `atexit()`.)

Exemple pratique 5

Comptage des caractères

Le programme présenté dans cette nouvelle section pratique, `count ch`, ouvre le fichier texte spécifié, et compte le nombre d’occurrences de chaque caractère rencontré. Tous les caractères standards du clavier sont pris en compte comme les majuscules et minuscules, les chiffres, les espaces, et les marques de ponctuation. Les résultats apparaissent à l’écran. Ce programme illustre quelques techniques de programmation intéressantes et fournit une application utile. Vous pourrez récupérer les résultats dans un fichier à l’aide de l’opérateur de redirection (>) :

```
count_ch > results.txt
```

Cette commande va exécuter le programme et enregistrer ses données en sortie dans le fichier `results.txt` plutôt que les afficher à l’écran. Il suffira ensuite d’éditer le fichier ou de l’imprimer.

Listing Exemple pratique 5 : compte_chaine.c : pour compter les caractères d’un fichier

```
/* Compte le nombre d'occurrences de chaque caractère
   dans un fichier. */
#include <stdio.h>
#include <stdlib.h>
int file_exists(char *filename);
int main()
{
     char ch, source[80];
     int index;
     long count[127];
```

```
     FILE *fp;

     /* Lecture des noms de fichiers source et destination. */
     fprintf(stderr, "\nEntrez le nom du fichier source: ");
     lire_clavier(source,sizeof(source));

     /* Contrôle de l'existance du fichier source. */
     if (!file_exists(source))
     {
        fprintf(stderr, "\n%s n'existe pas.\n", source);
        exit(EXIT_FAILURE);
     }
     /* Ouverture du fichier. */
     if ((fp = fopen(source, "rb")) == NULL)
     {
        fprintf(stderr, "\nErreur d'ouverture %s.\n", source);
        exit(EXIT_FAILURE);
     }
     /* Initialisation des éléments du tableau. */
     for (index = 31; index < 127 ; index++)
        count[index] = 0;

     while ( 1 )
     {
        ch = fgetc(fp);
        /* Fin si fin de fichier */
        if (feof(fp))
            break;
        /* Ne compte que les caractères entre 32 et 126. */
        if (ch > 31 && ch < 127)
            count[ch]++;
      }
     /* Affichage des résultats. */
     printf("\nChar\t\tCount\n");
     for (index = 32; index < 127 ; index++)
        printf("[%c]\t%d\n", index, count[index]);
     /* Fermeture du fichier et sortie. */
     fclose(fp);
     return(EXIT_SUCCESS);
}
int file_exists(char *filename)
{
/* Renvoie TRUE si le fichier existe,
   sinon FALSE. */
   FILE *fp;
   if ((fp = fopen(filename, "r")) == NULL)
       return 0;
   else
   {
       fclose(fp);
       return 1;
   }
}
```

Analyse

Vous pourrez utiliser la fonction `file exists()` des lignes 51 à 63 dans d'autres programmes quoique la fonction `stat()` suffise amplement à cette tâche. Elle s'assure de l'existence du fichier dont elle reçoit le nom en argument en tentant de l'ouvrir en mode lecture (ligne 56). Elle renvoie `TRUE` si le fichier existe et `FALSE` dans le cas contraire.

Notez l'utilisation de la fonction `fprintf()` pour afficher les messages à l'écran plutôt que `printf()` comme en ligne 14, par exemple. `printf()` envoie en effet toujours ses données en sortie vers `stdout` et l'utilisateur ne voie apparaître aucun message si l'opérateur de redirection renvoie ces données dans un fichier. L'utilisation de `fprintf()` impose l'envoie des messages vers `stderr`, dont le contenu est toujours affiché à l'écran.

Pour terminer cette analyse, observez la façon dont on a utilisé la valeur numérique de chaque caractère comme index dans le tableau des résultats (lignes 40 et 41). L'élément `count[32]`, par exemple, stocke le nombre d'espaces rencontrés parce que la valeur numérique 32 représente ce caractère.

17

Manipulation de chaînes de caractères

Le texte, sous forme de chaînes de caractères, constitue une partie importante de beaucoup de programmes. Jusqu'ici, vous avez appris à faire des entrées-sorties de texte, et vous savez comment les chaînes de caractères sont conservées en mémoire. Pour leur manipulation, C dispose d'une grande variété de fonctions. Dans ce chapitre, nous allons étudier :

- La longueur d'une chaîne
- La copie et la concaténation de chaînes
- Les fonctions de comparaison de chaînes
- Les recherches dans une chaîne de caractères
- La conversion d'une chaîne de caractères
- Comment tester des caractères à l'intérieur d'une chaîne

Longueur d’une chaîne

Vous savez que, dans un programme C, une chaîne est une suite de caractères dont le début est repéré par un pointeur et la fin par un zéro binaire (le caractère `NULL`, codé ’`\0'`). On a souvent besoin de connaître la longueur d’une chaîne, c’est-à-dire le nombre de caractères se trouvant entre le premier caractère et le zéro terminal. On utilise pour cela la fonction de bibliothèque standard `strlen()`, dont le prototype se trouve dans string.h :

```
size_t strlen(char *str);
```

Sans doute vous posez-vous des questions au sujet de ce curieux type : `size t.` Il est défini dans string.h comme étant un `unsigned`. La fonction `strlen()` renvoie donc un entier non signé. Ce type est utilisé pour la plupart des fonctions traitant des caractères. Souvenez-vous qu’il équivaut à `unsigned`.

L’argument passé à `strlen()` est un pointeur vers la chaîne de caractères dont vous voulez connaître la longueur. Vous récupérez le nombre de caractères réels, c’est-à-dire terminateur (`\0`) non compris. Le programme du Listing 17.1 montre un exemple d’utilisation de `strlen()`.

Listing 17.1 : Utilisation de la fonction strlen() pour connaître la longueur d’une chaîne de caractères

```
/* Utilisation de la fonction strlen(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
     size_t length;
     char buf[80];

    while (1)
    { puts("\nTapez une ligne de texte (une ligne vierge \
pour terminer).");
      lire_clavier(buf, sizeof(buf));

      length = strlen(buf);

      if (length != 0)
         printf("\nLa longueur de cette ligne est de %u \
              caractères.", length);
      else
         break;
    }
    exit(EXIT_SUCCESS);
}
```



```
Tapez une ligne de texte (une ligne vierge pour terminer).
Portez ce vieux whisky au juge blond qui fume.
La longueur de cette ligne est de 46 caractères.
Tapez une ligne de texte (une ligne vierge pour terminer).
```

Analyse

Aux lignes 13 et 14, on affiche un message et on lit une chaîne de caractères dans `buf`. À la ligne 16, on appelle `strlen()` qui permet de récupérer la longueur de la chaîne dans la variable `length`. Il ne reste plus alors qu'à l'afficher, ce qu'on fait à la ligne 19.

Copie de chaînes de caractères

Il existe trois fonctions pour copier des chaînes de caractères. Étant donné la façon dont elles sont traitées par C, on ne peut pas simplement faire une copie en assignant le contenu d'une variable à une autre, comme cela se pratique dans d'autres langages. On doit explicitement copier les caractères constitutifs de la première chaîne dans les emplacements mémoire affectés à la seconde. Les fonctions de recopie sont : `strcpy()`, `strncpy()` et `strdup()`. Si vous connaissez la taille de la zone à recopier, vous pouvez également utiliser `memcpy()` qui est à priori encore plus performante. Lorsqu'on appelle des fonctions de traitement de chaînes de caractères dans un programme, il ne faut pas oublier d'inclure le fichier d'en-tête string.h.

La fonction *strcpy()*

La fonction de bibliothèque `strcpy()` copie une chaîne entière dans une zone de mémoire. Son prototype est :

```
char *strcpy(char *destination, char *source);
```

Le caractère terminal `\0` est, lui aussi, recopié. La fonction renvoie un pointeur vers la nouvelle chaîne, `destination`.

Avant d'utiliser `strcpy()`, il faut allouer assez de place pour la chaîne destinataire, car `strcpy()` recopie systématiquement la totalité de la chaîne source. Le Listing 17.2 illustre l'utilisation de `strcpy()`.



Lorsqu'un programme utilise `malloc()` pour allouer de la mémoire, il est bon de libérer la mémoire acquise avant de terminer le programme en appelant la fonction `free()`. Nous étudierons cette dernière fonction au Chapitre 20.

Listing 17.2 : Avant d'appeler strcpy(), vous devez allouer assez de mémoire pour la chaîne destinataire

```
1:   /* Démonstration de strcpy(). */
2:
3:   #include <stdlib.h>
4:   #include <stdio.h>
5:   #include <string.h>
6:
7:   char source[] = "Une chaîne de caractères.";
8:
9:   int main()
10:  {
11:      char dest1[80];
12:      char *dest2;
13:
14:      printf("\nsource: %s", source);
15:
16:      /* Copier vers dest1 est correct parce que dest1 pointe
17:         vers une zone de 80 octets. */
18:
19:      strcpy(dest1, source);
20:      printf("\ndest1:  %s", dest1);
21:
22:      /* Pour copier vers dest2 vous devez allouer de la place.*/
23:
24:      dest2 = malloc(strlen(source) +1);
25:      strcpy(dest2, source);
26:      printf("\ndest2:  %s\n", dest2);
27:
28:      /* Faire une copie dans une zone non allouée est
29:         suicidaire. L'instruction suivante pourrait causer
30:         de sérieux problèmes :
31:         strcpy(dest3, source); */
32:      exit(EXIT_SUCCESS);
33:  }
```



```
source: Une chaîne de caractères.
dest1:  Une chaîne de caractères.
dest2:  Une chaîne de caractères.
```

Analyse

Il n'y a pas grand-chose à dire de ce programme. L'inclusion de stdlib.h est nécessaire, car on y trouve le prototype de `malloc()`. On remarquera cependant l'allocation dynamique de mémoire à la ligne 24, dont la longueur est déterminée par la longueur de la chaîne à recopier. Attention à ne pas "décommenter" les instructions des lignes 28 à 33 !

La fonction *strncpy()*

Cette fonction est similaire à `strcpy()`, à ce détail près qu'un troisième argument permet de fixer à l'avance la longueur de la chaîne source à recopier. Son prototype est le suivant :

```
char *strncpy(char *destination, char *source, size_t n);
```

Les arguments `destination` et `source` sont des pointeurs vers les chaînes de destination et d'origine. La fonction copie, au plus, les `n` premiers caractères de la chaîne source. Si `strlen(source)` est inférieur à `n`, les positions suivantes seront garnies par des valeurs `NULL`, à concurrence d'un total de `n` caractères recopiés. Si `strlen(source)` est supérieur à `n`, aucun terminateur ne se trouvera placé dans la chaîne `destination`.

Le programme du Listing 17.3 montre l'utilisation de `strncpy()`.

Listing 17.3 : Utilisation de la fonction strncpy()

```
/* Utilisation de la fonction strncpy(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char dest[] = "..........................";
char source[] = "abcdefghijklmnopqrstuvwxyz";

int main()
{
    size_t n;

    while (1)
    {
      puts("Indiquez le nombre de caractères à copier (1-26)");
      scanf("%d", &n);

      if (n > 0 && n < 27)
         break;
    }

    printf("\nAvant strncpy destination = %s", dest);

    strncpy(dest, source, n);

    printf("\nAprès strncpy destination = %s\n", dest);
    exit(EXIT_SUCCESS);
}
```



```
Indiquez le nombre de caractères à copier (1-26)
17
Avant strncpy destination = ..........................
Après strncpy destination = abcdefghijklmnopq.........
```

Analyse

Aux lignes 13 à 20, on trouve une boucle `while` demandant à l'utilisateur de taper un nombre compris entre 1 et 26. On ne sort de la boucle que lorsque la valeur tapée est correcte. Notez qu'il aurait été plus élégant d'écrire :

```
size_t n=0;

do
{  puts("Indiquez le nombre de caractères à copier (1-26)");
   scanf("%d", &n);
} while (n < 1 || n > 26);
```



Il ne faut pas que le nombre de caractères copiés excède l'emplacement alloué à cet effet.

La fonction *strdup()*

Cette fonction est identique à `strcpy()`, sauf qu'elle effectue sa propre allocation de mémoire pour la chaîne destinataire par un appel implicite à `malloc()`. Son prototype est le suivant :

```
char *strdup(char *source);
```

L'argument `source` est un pointeur vers la chaîne de caractères à recopier. La fonction renvoie un pointeur vers la chaîne contenant la copie, ou `NULL` si la mémoire nécessaire n'est pas disponible. Le Listing 17.4 montre un exemple d'utilisation de `strdup()`. Si cette fonction n'est pas une fonction ANSI elle est néanmoins conforme à des standards tels que POSIX et BSD 4.3, ce qui est un gage de protabilité.

Listing 17.4 : Utilisation de la fonction strdup() pour copier une chaîne avec allocation automatique de mémoire

```
/* La fonction strdup(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = "C'est la chaîne source.";

int main()
{
    char *dest;

    if ((dest = strdup(source)) == NULL)
```

```
    {
        fprintf(stderr, "Erreur d'allocation mémoire.");
        exit(EXIT_FAILURE);
    }

    printf("Destination = %s\n", dest);
    exit(EXIT_SUCCESS);
}
```



```
Destination = C'est la chaîne source.
```

La fonction *memcpy()*

Cette fonction est similaire à `strncpy()` au niveau de son prototype. Elle s'en distingue par le fait qu'elle copie exactement le nombre d'octets indiqués dans le troisième argument, sans tenir compte d'un éventuel caractère nul de fin de chaîne. Son prototype est le suivant :

```
void *memcpy(char *destination, char *source, size_t n);
```

Les arguments `destination` et `source` sont des pointeurs vers les chaînes de destination et d'origine. La fonction copie exactement les `n` premiers caractères de la chaîne source. Le grand intérêt par rapport à strcpy() et strncpy() est sa rapidité car elle n'a pas à tester en interne la présence d'un caractère nul. Cette fonction sert principalement lorsque vous connaissez déjà la longueur de la chaîne destination, ce qui est d'ailleurs le cas après avoir réservé l'espace mémoire nécessaire à la recopie.

Le programme du Listing 17.5 montre l'utilisation de `memcpy()`.

Listing 17.5 : Utilisation de la fonction memcpy()

```
/* Utilisation de la fonction memcpy(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = "abcdefghijklmnopqrstuvwxyz";

int main()
{
    char *dest
    size_t n;

    n = strlen(source) + 1;
    dest = malloc(n * sizeof(*dest));
    if(dest == NULL)
```

Listing 17.5 : Utilisation de la fonction memcpy() ***(suite)***

```
16:         {
17:         fprintf(stderr, "Erreur d'allocation mémoire.");
18:         exit(EXIT_FAILURE);
19:       }
20:
21:       memcpy(dest, source, n);
22:
23:       printf("\nAprès memcpy destination = %s\n", dest);
24:       exit(EXIT_SUCCESS);
25:   }
```



```
Après memcpy destination = abcdefghijklmnopqrstuvwxyz
```

Analyse

Nous avons besoin de la taille de l'espace mémoire à réserver, taille qui est calculée ligne 13. La mémoire est réservée dès la ligne suivante. Lorsqu'il s'agit de recopier la chaîne `source`, il est à la fois inutile de recalculer la longueur de la chaîne (puisque nous la connaissons depuis la ligne 13) et inutile d'utiliser une fonction comme `strcpy()` qui va perdre du temps à rechercher le caractère nul de fin de chaîne. C'est donc `memcpy()` la fonction la plus adaptée ici.

Remarquez au passage que les lignes 11 à 21 auraient pu être remplacées par un simple `dest = strdup(source)`.

Concaténation de chaînes de caractères

Peut-être ce terme ne vous dit-il rien ? Concaténer deux objets, c'est les mettre bout à bout (lorsque c'est possible, bien sûr, ce qui est le cas pour les chaînes de caractères). La bibliothèque standard du C contient deux fonctions à cet usage : `strcat()` et `strncat()`. Toutes deux nécessitent l'inclusion du fichier string.h.

La fonction *strcat()*

Son prototype est :

```
char *strcat(char *str1, char *str2);
```

Cette fonction concatène les chaînes `str1` et `str2`, c'est-à-dire qu'elle ajoute une copie de `str2` à la suite de `str1`. Le programme du Listing 17.6 donne un exemple d'utilisation de `strcat()`.

Listing 17.6 : Utilisation de la fonction strcat() pour concaténer deux chaînes de caractères

```
/* La fonction strcat(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char str1[27] = "a";
char str2[2];

int main()
{
    int n;

    /* On met un caractère NULL à l'extrémité de str2[]. */

    str2[1] = '\0';

    for (n = 98; n < 123; n++)
    {
        str2[0] = n;
        strcat(str1, str2);
        puts(str1);
    }
    exit(EXIT_SUCCESS);
}
```

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

Analyse

Les codes ASCII des lettres *b* à *z* sont 98 à 122. Ce programme utilise ces codes ASCII pour donner une illustration de l'emploi de `strcat()`. La boucle `for` des lignes 17 à 22 assigne ces valeurs l'une après l'autre à `str2[0]`. Comme `str2[1]` contient déjà le terminateur (ligne 15), il en résulte un garnissage progressif illustré par la sortie écran de la ligne 21.

La fonction *strncat()*

Cette fonction de bibliothèque effectue une concaténation que vous pouvez limiter puisque le troisième argument en précise la portée. Le prototype est :

```
char *strncat(char *str1, char *str2, size_t n);
```

Si `str2` contient plus de `n` caractères, ses `n` premiers caractères sont ajoutés à l'extrémité de `str1`. Si `str2` contient moins de `n` caractères, toute la chaîne est ajoutée à l'extrémité de `str1`. Dans les deux cas, un terminateur est placé à la fin de `str1`. Vous devez allouer assez de place à `str1` pour la réunion des deux chaînes. Le programme du Listing 17.7 illustre l'utilisation de cette fonction.

Listing 17.7 : Utilisation de la fonction strncat() pour concaténer deux chaînes de caractères

```
/* La fonction strncat(). */

#include <stdio.h>
#include <string.h>

char str2[] = "abcdefghijklmnopqrstuvwxyz";

int main()
{
    char str1[27];
    int n;

    for (n=1; n < 27; n++)
    {
        strcpy(str1, "");
        strncat(str1, str2, n);
        puts(str1);
    }
    exit(EXIT_SUCCESS)
}
```

```
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

Analyse

Vous vous interrogez peut-être au sujet de l'instruction qui figure sur la ligne 15 : `strcpy(str1, "");`. Elle recopie une chaîne vide, c'est-à-dire ne contenant que le seul terminateur (`\0`). Il en résulte que le premier caractère de `str1`, `str1[0]`, est `NULL`. On aurait pu faire la même chose en écrivant : `str1[0] = 0;` ou `str1[0] = '\0';`.

Nous vous rappelons par ailleurs que si vous connaissez la longueur des deux chaînes de caractères, il est plus efficace d'utiliser `memcpy`(). Les trois lignes suivantes effectuent la même chose :

```
strcat(str1, str2);
strncat(str1, str2, n2);
memcpy(str1+n1, str2, n2+1);
```

n1 et n2 sont les longueurs des chaînes de caractères telles que renvoyées par la fonction `strlen()` (et non pas la taille de l'espace mémoire qui inclut lui le caractère nul de fin de chaîne).

Comparaison de deux chaînes de caractères

Lorsqu'on compare deux chaînes de caractères, c'est, le plus souvent, pour savoir si elles sont différentes. Si elles sont inégales, l'une d'elles est "inférieure" à l'autre. Cette "infériorité" est déterminée par la valeur des codes ASCII qui, par bonheur, respectent l'ordre alphabétique. les lettres majuscules (codes ASCII de 65 à 90) ont assez bizarrement des valeurs inférieures à leurs équivalents minuscules (codes ASCII de 97 à 122). La chaîne `"ZEBRA"` va donc être évaluée comme inférieure à la chaîne `"apple" par ces fonctions C`.

La bibliothèque C ANSI contient deux types de fonctions de comparaison : entre deux chaînes entières et entre deux chaînes sur une longueur prédéterminée.

Comparaison de deux chaînes entières : la fonction *strcmp()*

La fonction `strcmp()` compare deux chaînes de caractères, caractère par caractère. Son prototype se trouve dans string.h :

```
int strcmp(char *str1, *str2);
```

Les arguments `str1` et `str2` pointent sur les deux chaînes à comparer. La fonction renvoie les valeurs indiquées par le Tableau 17.1, et le programme du Listing 17.8 donne un exemple d'utilisation.

Tableau 17.1 : Valeurs renvoyées par strcmp()

Valeur de retour	*Signification*
`< 0`	`str1 < str2`
`= 0`	`str1 = str2`
`> 0`	`str1 > str2`

Listing 17.8 : Utilisation de strcmp() pour comparer deux chaînes de caractères

```
/* La fonction strcmp(). */

#include <stdio.h>
#include <string.h>

int main()
{
    char str1[80], str2[80];
```

```
 9:       int x;
10:
11:       while (1)
12:       {
13:           /* Lecture au clavier de deux chaînes de caractères. */
14:
15:           printf("\n\nTapez la première chaîne (Entrée pour \
16:                   terminer) : ");
17:           lire_clavier(str1, sizeof(str1));
18:
19:           if (strlen(str1) == 0)
20:               break;
21:
22:           printf("\nTapez la seconde chaîne : ");
23:           lire_clavier(str2, sizeof(str2));
24:
25:           /* Comparaison des deux chaînes et affichage du résultat.*/
26:
27:           x = strcmp(str1, str2);
28:
29:           printf("\nstrcmp(%s,%s) renvoie %d", str1, str2, x);
30:       }
31:       exit(EXIT_SUCCESS);
32:  }
```

```
Tapez la première chaîne (Entrée pour terminer) : Première chaîne
Tapez la seconde chaîne : Seconde chaîne
strcmp(Première chaîne,Seconde chaîne) renvoie -1

Tapez la première chaîne (Entrée pour terminer) : abcdefgh
Tapez la seconde chaîne : abcdefgh
strcmp(abcdefgh,abcdefgh) renvoie 0

Tapez la première chaîne (Entrée pour terminer) : zoologue
Tapez la seconde chaîne : abricot
strcmp(zoologue,abricot) renvoie 1

Tapez la première chaîne (Entrée pour terminer) :
```

Analyse

L'utilisateur est invité à taper ses deux chaînes de caractères (lignes 15, 17, 22 et 23), le résultat est affiché par le `printf()` de la ligne 19. Faites quelques essais avec ce programme, en tapant deux fois la même chaîne, une fois en minuscules, l'autre en majuscules, par exemple.

Comparaison partielle de deux chaînes de caractères : la fonction *strncmp()*

La fonction de bibliothèque `strncmp()` compare un nombre donné de caractères pris dans deux chaînes de caractères. Voici son prototype :

```
int strncmp(char *str1, *str2, size_t n);
```

Les arguments `str1` et `str2` pointent sur les deux chaînes à comparer et `n` indique le nombre de caractères à comparer. La fonction renvoie les valeurs indiquées par le Tableau 17.1 ; le programme du Listing 17.9 donne un exemple d'utilisation.

Listing 17.9 : Comparaison partielle de deux chaînes de caractères

```
/*  La fonction strncmp(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char str1[] = "Voici la première chaîne.";
char str2[] = "Voici la seconde chaîne.";

int main()
{
    size_t n, x;

    puts(str1);
    puts(str2);

    while (1)
    {
        puts("\n\nTapez le nombre de caractères à comparer, \
0 pour terminer.");
        scanf("%d", &n);
        if (n <= 0)
            break;

        x = strncmp(str1, str2, n);

        printf("\nComparaison de %d caractères. strncmp() \
renvoie %d.", n, x);
    }
    exit(EXIT_SUCCESS);
}
```

```
Voici la première chaîne.
Voici la seconde chaîne.

Tapez le nombre de caractères à comparer, 0 pour terminer.
9
Comparaison de 9 caractères. strncmp() renvoie 0.

Tapez le nombre de caractères à comparer, 0 pour terminer.
```

```
12
Comparaison de 12 caractères. strncmp() renvoie -1.

Tapez le nombre de caractères à comparer, 0 pour terminer.
0
```

Analyse

Le programme compare les deux chaînes respectivement définies aux lignes 6 et 7. Les lignes 13 et 14 affichent les chaînes sur l'écran afin que l'utilisateur puissent facilement se repérer. La boucle `while` des lignes 16 à 28 autorise plusieurs essais. On en sort lorsque l'utilisateur tape 0 (lignes 21 et 22). Le résultat est affiché à la ligne 26.

Comparaison de deux chaînes en ignorant leur casse

La bibliothèque C ANSI ne fournit malheureusement aucune fonction de comparaison de chaînes qui ne tienne pas compte de la casse. Cependant, vous trouverez les fonctions `strcasecmp()` et `strncasecmp()` sur les systèmes d'exploitations qui respectent la norme POSIX (comme Windows et Linux). Certains compilateurs C proposent également leurs propres fonctions "maison" pour cette opération. Symantec utilise la fonction `strcmpl()`, Microsoft fait appel à la fonction `stricmp()` et Borland propose `strcmpi()` et `stricmp()`. Consultez le manuel de référence de votre bibliothèque pour connaître la fonction spécifique de votre compilateur. Lorsque vous utilisez ce type de fonction, les deux chaînes `Smith` et `SMITH` apparaissent identiques. Modifiez la ligne 27 du Listing 17.8 avec la fonction de comparaison appropriée (qui ignore la casse) en fonction de votre compilateur et testez ce programme de nouveau.

Recherche dans une chaîne de caractères

La bibliothèque C contient six fonctions effectuant des recherches dans une chaîne de caractères. Toutes demandent l'inclusion de string.h.

La fonction *strchr()*

La fonction `strchr()` recherche la première occurrence d'un caractère particulier. Son prototype est :

```
char *strchr(char *str, int ch);
```

La recherche s'effectue de la gauche vers la droite, c'est-à-dire dans l'ordre croissant des positions. Elle s'arrête dès qu'une égalité est trouvée ou que l'on parvient au bout de la chaîne. En cas de réussite, la fonction renvoie un pointeur vers le caractère cherché. Elle renvoie `NULL` en cas d'échec.

Pour obtenir la position du caractère trouvé (lorsque c'est le cas), il suffit de faire la différence entre la valeur renvoyée et l'adresse du début de la chaîne. Le programme du Listing 17.10 montre comment on peut opérer. Souvenez-vous que le premier caractère d'une chaîne occupe la position 0. Comme beaucoup d'autres fonctions de C qui s'appliquent aux chaînes, `strchr()` différencie les majuscules des minuscules. Elle indiquera, par exemple, que le caractère `F` est absent de la chaîne `raffle`.

Listing 17.10 : Utilisation de strchr() pour rechercher la position d'un caractère dans une chaîne

```
 1:   /* Recherche de la position d'un caractère dans une chaîne
 2:      avec strchr(). */
 3:   #include <stdio.h>
 4:   #include <stdlib.h>
 5:   #include <string.h>
 6:   int main()
 7:   {
 8:       char *loc, buf[80];
 9:       int ch;
10:
11:       /* Taper la chaîne de caractères et le caractère. */
12:
13:       printf("Tapez la chaîne de caractères : ");
14:       lire_clavier(buf, sizeof(buf));
15:       printf("Tapez le caractère à chercher : ");
16:       ch = getchar();
17:
18:       /* effectuer la recherche. */
19:
20:       loc = strchr(buf, ch);
21:
22:       if (loc == NULL)
23:           printf("On n'a pas trouvé le caractère %c.", ch);
24:       else
25:           printf("Le caractère %c a été trouvé à la position \
26:   %d.", ch, loc-buf);
27:       exit(EXIT_SUCCESS);
28:   }
```

```
Tapez la chaîne de caractères : Il était un petit navire
Tapez le caractère à chercher : p
Le caractère p a été trouvé à la position 12.
```

Analyse

C'est l'appel à `strchr()` de la ligne 20 qui effectue la recherche. Le test de la ligne 22 permet de choisir entre les deux messages à afficher, selon le résultat de la recherche. En cas de réussite, la position du caractère trouvé est déterminée par la soustraction effectuée à la ligne 26.

La fonction *strrchr()*

`strrchr()` est analogue à `strchr()`, à cet important détail près : au lieu de rechercher la première occurrence d'un caractère spécifié dans une chaîne, elle recherche sa *dernière* occurrence. Son prototype est :

```
chr *strrchr(char *str, int ch);
```

Cette fonction renvoie un pointeur vers la dernière occurrence du caractère spécifié, ou `NULL` si ce caractère ne figure pas dans la chaîne. Pour tester son fonctionnement, il suffit de modifier la ligne 20 dans le Listing 17.10 en remplaçant `strchr()` par `strrchr()`.

La fonction *strcspn()*

La fonction de bibliothèque `strcspn()` recherche la première occurrence dans une chaîne, de l'un des caractères d'une seconde chaîne. Son prototype est le suivant :

```
char *strcspn(char *str1, char *str2);
```

La fonction commence par s'attaquer au premier caractère de `str1` en cherchant s'il est égal à l'un des caractères de `str2`. Si ce n'est pas le cas, elle passe au deuxième caractère de `str2` et ainsi de suite. Il est important de se souvenir que la fonction ne recherche pas la chaîne `str2`, mais seulement les caractères qu'elle contient. Lorsqu'elle trouve une égalité, elle renvoie un pointeur vers l'emplacement du caractère trouvé dans `str1`. En cas d'échec, elle renvoie `strlen(str1)`, indiquant ainsi que la correspondance n'existe qu'avec le terminateur de `str1`. Le programme du Listing 17.11 montre un exemple d'utilisation de `strcspn()`.

Listing 17.11 : Recherche d'un caractère parmi plusieurs dans une chaîne de caractères avec strcspn()

```
/* Recherche avec strcspn(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char  buf1[80], buf2[80];
    size_t loc;

    /* Entrée des chaînes de caractères. */
    printf("Tapez la chaîne de caractères dans laquelle on \
 cherchera :\n");
    lire_clavier(buf1, sizeof(buf1));
    printf("Tapez la chaîne de caractères contenant les \
 caractères à chercher :\n");
    lire_clavier(buf2, sizeof(buf2));
```

Listing 17.11 : Recherche d'un caractère parmi plusieurs dans une chaîne de caractères avec strcspn() *(suite)*

```
18:
19:       /* Effectuer la recherche. */
20:       loc = strcspn(buf1, buf2);
21:
22:       if (loc ==  strlen(buf1))
23:           printf("On n'a trouvé aucune correspondance.");
24:       else
25:           printf("La première correspondance a été trouvée \
26:    à la position %d.\n", loc);
27:       exit(EXIT_SUCCESS);
28:   }
```



```
Tapez la chaîne de caractères dans laquelle on cherchera :
le chat de la voisine
Tapez la chaîne de caractères contenant les caractères à chercher :
bord
La première correspondance a été trouvée à la position 8.
```

Analyse

Le programme ressemble à celui du Listing 17.10, mais au lieu de rechercher l'occurrence d'un seul caractère, on recherche, cette fois, les possibilités d'occurrence d'un des caractères d'une chaîne. Ici, le premier caractère commun aux deux chaînes est le *d* de "de" qui correspond au *d* de "bord". On remarquera le test qui permet (ligne 22) d'afficher qu'aucune correspondance n'a été trouvée. Il est différent du test habituel. La simplicité du programme n'appelle guère d'autres commentaires.

La fonction *strspn()*

Cette fonction s'apparente à la précédente, comme nous allons le voir dans le paragraphe suivant. Son prototype est :

```
size_t strspn(char *str1, char *str2);
```

La fonction `strspn()` recherche dans `str1` la position du premier caractère n'ayant pas d'équivalent dans `str2` et renvoie cette position, ou `NULL` si aucune correspondance n'est découverte. Le programme du Listing 17.12 montre un exemple d'utilisation.

Listing 17.12 : Recherche du premier caractère n'ayant pas de correspondance avec strspn()

```
/* Recherche avec strspn(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main()
{
    char  buf1[80], buf2[80];
    size_t loc;

    /* Entrée des deux chaînes. */
    printf("Tapez la chaîne de caractères dans laquelle \
on cherchera :\n");
    lire_clavier(buf1, sizeof(buf1);
    printf("Tapez la chaîne de caractères contenant les \
caractères à chercher :\n");
    lire_clavier(buf2, sizeof(buf2);
    /* Effectuer la recherche. */

    loc = strspn(buf1, buf2);

    if (loc ==  0)
        printf("On n'a trouvé aucune correspondance.\n");
    else
        printf("Il y a correspondance jusqu'à la position %d.\n",
                loc-1);
        exit(EXIT_SUCCESS);
}
```

Le fonctionnement de cette fonction n’étant pas évident, voici trois essais qui permettront d’y voir plus clair :

```
Tapez la chaîne de caractères dans laquelle on cherchera :
Le chat de la voisine
Tapez la chaîne de caractères contenant les caractères à chercher :
Le chat de la cousine
Il y a correspondance jusqu'à la position 13.

Tapez la chaîne de caractères dans laquelle on cherchera :
Le chat de la voisine
Tapez la chaîne de caractères contenant les caractères à chercher :
mur
On n'a trouvé aucune correspondance.

Tapez la chaîne de caractères dans laquelle on cherchera :
Le chat de la cousine
Tapez la chaîne de caractères contenant les caractères à chercher :
Le chat de la voisine
Il y a correspondance jusqu'à la position 15.
```

Analyse

La structure du programme étant identique à celle de l’exemple précédent, il est inutile de répéter les explications qui ont été données à cette occasion. En revanche, dans les exemples

ci-avant, on voit que dans le premier cas, la première lettre de `str1` n'ayant pas d'équivalent dans `str2` est le *v* qui occupe la position 13. Dans le deuxième cas, `str2` ne contient aucune des lettres de `str1`. Enfin, dans le troisième, c'est le *u* de "cousine" qui n'a aucune correspondance dans `str2`.

La fonction *strpbrk()*

La fonction de bibliothèque `strpbrk()` ressemble à la fonction `strcspn()`. Elle recherche, elle aussi, la première occurrence, dans une chaîne, d'un des caractères d'une seconde chaîne ; mais, en plus, elle inclut le terminateur `\0` dans la recherche. Son prototype est :

```
char * strpbrk(char *str1, char *str2)
```

Elle renvoie un pointeur vers le premier caractère de `str1` qui correspond à un caractère quelconque de `str2` ; `NULL` si aucune correspondance n'est trouvée. Comme nous venons de le dire pour `strchr()`, on peut obtenir la position de ce caractère en soustrayant de la valeur de retour (si elle est différente de `NULL`, bien sûr) l'adresse du début de `str1`.

La fonction *strstr()*

La dernière, et peut-être la plus utile, de nos fonctions de manipulation de caractères est `strstr()`. Elle recherche la première occurrence d'une chaîne à l'intérieur d'une autre. Cette recherche s'applique à la chaîne complète, et non aux caractères qui la composent. Son prototype est :

```
char *strstr(char *str1, char *str2);
```

Elle retourne un pointeur vers la première occurrence de `str2` dans `str1`, ou `NULL` si aucune correspondance n'est trouvée. Si la longueur de `str2` est égale à zéro, la fonction retourne l'adresse du début de `str1`. Comme nous venons de le dire plus haut, on peut obtenir la position de ce caractère en soustrayant de la valeur de retour (si elle est différente de `NULL`) l'adresse du début de `str1`. Le programme du Listing 17.13 montre un exemple d'utilisation.

Listing 17.13 : Utilisation de la fonction strstr() pour rechercher une chaîne dans une autre

```
/* Recherche avec strstr(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
```

```
{
    char *loc, buf1[80], buf2[80];

    /* Acquisition des deux chaînes. */
    printf("Tapez la chaîne de caractères dans laquelle \
           s'effectuera la recherche :\n");
    lire_clavier(buf1, sizeof(buf1));
    printf("Tapez la chaîne à rechercher :\n");
    lire_clavier(buf2, sizeof(buf2));

    /* Effectuer la recherche. */

    loc = strstr(buf1, buf2);

    if (loc ==  NULL)
        printf("Pas de correspondance.");
    else
        printf("%s a été découvert à la position %d.", buf2, \
               loc-buf1);
    exit(EXIT_SUCCESS);
}
```

L'utilisation de cette fonction étant simple, nous ne donnerons qu'un seul exemple :

```
Tapez la chaîne de caractères dans laquelle s'effectuera la recherche :
le chat de la voisine
Tapez la chaîne à rechercher :
vois
vois a été découvert à la position 14.
```



À faire

Souvenez-vous que, pour la plupart des fonctions de traitement de chaînes, il existe des fonctions équivalentes permettant de spécifier le nombre des caractères sur lesquels doit porter la manipulation. Leur nom s'écrit généralement sous la forme `strnxxx()`.

Lorsque vous voulez recopier une chaîne de caractères dont vous connaissez la longueur, préférez `memcpy()`.

Analyse

Pas de commentaire particulier ; on se reportera éventuellement à ceux des trois fonctions précédentes.

Conversions de chaînes

Il existe deux fonctions pour modifier la casse d'un caractère :

```
char *tolwr(char c);
char *toupr(char c);
```

Ces deux fonctions nécessitent l'inclusion du fichier d'en-tête `ctype.h`. La première convertit le caractère fourni en argument de majuscules en minuscules et la seconde fait le contraire. Les caractères accentués subsistent tels quels. Il n'existe pas de fonction d'un standard répandu qui convertisse une chaîne de caractères.

Listing 17.14 : Conversion de casse avec les fonctions tolwr() et toupr()

```
/* Les fonctions de conversion de casse strlwr()et strupr(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int main()
{
    char buf[80];
    int i;

    while (1)
    {
       puts("Tapez une ligne de texte ou Entrée pour terminer.");
       lire_clavier(buf, sizeof(buf));

       if (strlen(buf) == 0)
          break;

       for(i=0; i<strlen(buf); i++) buf[i] = tolower(buf[i]);
       puts(buf);

       for(i=0; i<strlen(buf); i++) buf[i] = toupper(buf[i]);
       puts(buf);
    }
    exit(EXIT_SUCCESS);
}
```

Voici un exemple montrant ce qui se passe avec des caractères accentués :

```
Tapez une ligne de texte ou Entrée pour terminer.
L'oeil était dans la tombe et regardait Caïn.
l'oeil était dans la tombe et regardait caïn.
L'OEIL éTAIT DANS LA TOMBE ET REGARDAIT CAïN.
Tapez une ligne de texte ou Entrée pour terminer.
```

Fonctions de conversion d'une chaîne de caractères en nombre

Il existe plusieurs fonctions permettant de convertir une chaîne de caractères en sa représentation numérique (ce qui permet de l'utiliser dans des calculs). Par exemple, la chaîne "123" peut être convertie en un entier `int` dont la *valeur* sera égale à 123. Leurs prototypes se trouvent dans stdlib.h.

La fonction *strtol()*

Cette fonction convertit une chaîne de caractères en une valeur de type `long int`. Son prototype est :

```
long int strtol(const char *ptr, char **endptr, int base);
```

Elle convertit la chaîne pointée par `ptr` en un entier signé selon la base indiquée en troisième argument. Généralement, vous souhaiterez convertir la chaîne en base 10. Pour cela, vous mettrez NULL en deuxième argument et 10 en troisième. La fonction `strtol()` reconnaît, outre les chiffres 0 à 9, les caractères + et –. La conversion débute en tête de la chaîne et se poursuit jusqu'à la rencontre d'un caractère ne faisant pas partie de l'ensemble des caractères reconnus. Si la fonction ne reconnaît aucun de ces caractères, elle renvoie 0. Pour faire la différence entre la conversion de la chaîne "0" (ou équivalente comme par exemple "-00") et un problème de conversion qui renverrait également 0, vous devez tester la variable errno que vous aurez mis auparavant à 0. Le Tableau 17.2 donne quelques exemples :

Tableau 17.2 : Conversions effectuées par strtol()

Chaîne	*Valeur renvoyée*
`"157"`	`157`
`" 1.8"`	`1.8`
`"+50x"`	`50`
`"douze"`	`0`
`"x506"`	`0`

Dans les trois derniers exemples, on vérifie que la rencontre d'un caractère non numérique et autre qu'un signe met fin à la conversion.

Le programme du Listing 17.14 montre un exemple d'utilisation.

Listing 17.14 : Utilisation de la fonction strtol() pour convertir une chaîne de caractères

```
/* convertion de chaîne en long avec strtol(). */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    char nombre1[] = "123";
    char nombre2[] = "-00";
    char nombre3[] = "douze";
    long n;

    errno = 0;
    n = strtol(nombre1, NULL, 10);
    if(errno)
        printf("%s n'est pas un nombre\n", nombre1);
    else
        printf("%s vaut %d\n", nombre1, n);

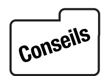
    errno = 0;
    n = strtol(nombre2, NULL, 10);
    if(errno)
        printf("%s n'est pas un nombre\n", nombre2);
    else
        printf("%s vaut %d\n", nombre2, n);

    errno = 0;
    n = strtol(nombre3, NULL, 10);
    if(errno)
        printf("%s n'est pas un nombre\n", nombre3);
    else
        printf("%s vaut %d\n", nombre3, n);

    exit(EXIT_SUCCESS);
}
```



```
123 vaut 123
-00 vaut 0
douze n'est pas un nombre
```



À faire

Initialiser `errno` avant chaque appel à `strtol()`.

Vérifier par la valeur de `errno` qu'il n'y a pas eu d'erreur lors de l'appel à `strtol()`.

À ne pas faire

Utiliser les fonctions `atoi()`, `atol()` et `atoll()` qui ne font pas la différence entre une chaîne contenant 0 et une chaîne impossible à convertir.

Analyse

La conversion des chaînes de caractères en long s'effectue à chaque fois de la même manière avec le deuxième argument de `strtol()` à NULL et le troisième à 10. Il faut initialiser la variable externe `errno` à 0 avant chaque appel. Si la chaîne ne contient pas de nombre, `strtol()` modifie `errno`. C'est ainsi que l'on fait la différence entre une chaîne contenant 0 et une chaîne qui ne contient pas de nombre.

La fonction `atoi()` effectue également la conversion mais vous devez l'éviter car contrairement à `strtol()`, elle ne fait pas la différence entre une chaîne contenant 0 et une chaîne ne pouvant être convertie.

La fonction *strtoll()*

Cette fonction fait le même travail que `strtol()`, mais, cette fois, le résultat de la conversion est de type `long long int`. Son prototype est :

```
long long int strtoll(const char *ptr, char **endptr, int base);
```

La fonction *strtoul()*

Cette fonction fait le même travail que `strtol()`, mais, cette fois, le résultat de la conversion est non signé, de type `unsigned long int`. Son prototype est :

```
unsigned long int strtoul(const char *ptr, char **endptr, int base);
```

La fonction *strtoull()*

Cette fonction fait le même travail que `strtoll()`, mais, cette fois, le résultat de la conversion est non signé, de type `unsigned long long int`. Son prototype est :

```
unsigned long long int strtoull(const char *ptr, char **endptr, int base);
```

La fonction *strtod()*

Cette fonction convertit une chaîne de caractères en une valeur numérique de type `double`. Son prototype est :

```
double strtod(char *ptr, char **endptr);
```

Elle convertit la chaîne pointée par `ptr` en un nombre flottant double précision. Comme avec `strtol()`, le second argument est peu utile et peut être mis à NULL. À cet effet, cette fonction reconnaît, outre les chiffres 0 à 9, les caractères + et –, les caractères *E* et *e* (exposant) et admet un ou plusieurs blancs en tête. La conversion débute au début de la chaîne et se poursuit jusqu'à la rencontre d'un caractère ne faisant pas partie de l'ensemble des caractère reconnus. Si la fonction ne reconnaît aucun de ces caractères, elle renvoie 0. Pour faire la différence entre la conversion de la chaîne "0" (ou équivalente comme par exemple "-0.0") et un problème de conversion qui renverrait également 0, vous devez tester la variable errno que vous aurez mis auparavant à 0. Le Tableau 17.3 donne quelques exemples :

Tableau 17.3 : Conversions effectuées par strtod()

Chaîne	***Valeur renvoyée***
`"12"`	12.000000
`" 0.123"`	– 0.123000
`"125E+3"`	123000.000000
`"123.1e 5"`	0.001231

Listing 17.15 : Utilisation de la fonction strtod()pour convertir une chaîne de caractères en une valeur numérique de type double

```
/* Démonstration de strtod(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main()
{
     char buf[80];
    double d;

    while (1)
    {   printf("\nTapez la chaîne de caractères à convertir \
(Entrée pour terminer): ");
        lire_clavier(buf, sizeof(buf));

        if (strlen(buf) == 0)
            break;

        errno = 0;
        d = strtod(buf, NULL);
```

```
22:            if(errno)
23:            {
24:                printf("Pas de valeur à convertir dans la chaîne\n");
25:            } else {
26:                printf("Valeur convertie : %f.", d);
27:            }
28:        }
29:        exit(EXIT_SUCCESS);
30:   }
```

Voici quelques exemples de conversions :

```
Tapez la chaîne de caractères à convertir (Entrée pour terminer):
123.45
Valeur convertie : 123.450000.
Tapez la chaîne de caractères à convertir (Entrée pour terminer):
-67.
Valeur convertie : -67.000000.
Tapez la chaîne de caractères à convertir (Entrée pour terminer):
abc
Pas de valeur à convertir dans la chaîne
Tapez la chaîne de caractères à convertir (Entrée pour terminer):
0
Valeur convertie : 0.000000.
Tapez la chaîne de caractères à convertir (Entrée pour terminer):
```

Analyse

La boucle `while` des lignes 12 à 28 permet de faire plusieurs essais. Aux lignes 14 et 15, on demande à l'utilisateur de taper une valeur. À la ligne 17, on regarde s'il a simplement tapé Entrée, auquel cas, le programme se termine. La variable externe errno est mise à zéro ligne 20 pour tester une éventuelle erreur ligne 22. La chaîne de caractères tapée est convertie en un nombre flottant à la ligne 21, puis affichée à la ligne 24 ou 26 selon qu'il y avait quelque chose à convertir (ligne 26) ou non (ligne 24)..

La fonction *strtof()*

Cette fonction fait le même travail que `strtod()`, mais, cette fois, le résultat de la conversion est de type `float`. Son prototype est :

```
float strtoull(const char *ptr, char **endptr);
```

Cette fonction est supportée par les compilateurs reconnaissant la norme ISO C99 mais pas les normes ISO C précédentes (ANSI et C89). Avec le compilateur gcc, vous devrez ainsi lui indiquer l'option `std=c99`. Il est également possible d'indiquer la macro suivante pour spécifier que votre code est conforme à cette norme :

```
#define _ISOC99_SOURCE
```

La fonction *sprintf()*

Cette fonction est l'inverse de toutes les fonctions précédentes. Elle convertit des nombres (et des chaînes) en une chaîne de caractères. Son prototype est :

```
int sprintf(char *str, const char *format, ...);
```

Cette fonction, initialement absente du standard ANSI, est largement supportée car conforme aux normes C89 et C99. Elle fonctionne comme `fprintf()` que nous avons vue aux Chapitres 14 et 16, à ceci près que le premier argument est un pointeur vers une chaîne de caractères et non un descripteur de flux. Par conséquent, vous pouvez indiquer l'adresse d'un espace mémoire et `sprintf()` le remplira d'une chaîne en convertissant les données suivant le format indiqué en deuxième argument. Dans les exemples suivants, chacune des deux lignes a l'effet inverse (on suppose que les pointeurs ont été correctement initialisés vers des espaces mémoires de taille suffisante).

Exemple 1

```
n = strtol("123", NULL, 10);
sprintf(str, "%d", 123);
```

Exemple 2

```
d = strtod("11.23", NULL);
sprintf(str, "%f", 11.23);
```

Fonctions de test de caractères

Le fichier d'en-tête ctype.h contient les prototypes d'un certain nombre de fonctions de test de caractères qui renvoient vrai ou faux, selon que le caractère testé remplit ou non la condition indiquée. Par exemple : "Est-ce une lettre ou un chiffre ?" Ces fonctions sont en réalité des macros. Ce n'est qu'au Chapitre 21 que vous saurez tout sur les macros et, à ce moment, vous pourrez regarder dans ctype.h comment sont définies les fonctions que nous allons maintenant étudier. Toutes ces macros ont le même prototype :

```
int isxxxx(int ch);
```

où `xxx` est le nom d'une macro particulière et `ch`, le caractère à tester. Le Tableau 17.4 donne la liste de ces macros.

Vous pouvez faire des choses très intéressantes à l'aide de ces fonctions. Par exemple, la fonction `get int()` du Listing 17.17 lit une chaîne de caractères représentant un entier sur `stdin`, et renvoie une valeur numérique de type `int`. Elle ignore les espaces en tête et renvoie 0 si le premier caractère rencontré n'est pas un chiffre.

Tableau 17.4 : Liste des macros isxxxx()

Macro	*Action*
`isalnum()`	Renvoie vrai si `ch` est une lettre ou un chiffre
`isalpha()`	Renvoie vrai si `ch` est une lettre
`isascii()`	Renvoie vrai si `ch` est un caractère ASCII standard (compris entre 0 et 127)
`iscntrl()`	Renvoie vrai si `ch` est un caractère de contrôle
`isdigit ()`	Renvoie vrai si `ch` est un chiffre
`isgraph ()`	Renvoie vrai si `ch` est un caractère imprimable (autre qu'un espace)
`islower ()`	Renvoie vrai si `ch` est une lettre minuscule (bas de casse)
`isprint ()`	Renvoie vrai si `ch` est un caractère imprimable (espace compris)
`ispunct ()`	Renvoie vrai si `ch` est un caractère de ponctuation
`isspace ()`	Renvoie vrai si `ch` est un séparateur (espace, tabulation, tabulation verticale, à la ligne, saut de page ou retour chariot)
`isupper ()`	Renvoie vrai si `ch` est une lettre majuscule (capitale)
`isxdigit ()`	Renvoie vrai si `ch` est un chiffre hexadécimal (0 à 9 et A à F)

Listing 17.17 : Utilisation des macros isxxx() pour implémenter une fonction de lecture au clavier d'un nombre entier

```
/* Utilisation des macros de test de caractères pour réaliser
   une fonction lisant un entier au clavier */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int get_int(void);

int main()
{
    int x;
    x =  get_int();
    printf("Vous avez tapé : %d.\n", x);
    exit(EXIT_SUCCESS);
}

int get_int(void)
{
    int ch, i, sign = 1;

    /* Ignorer les espaces en tête. */

    while (isspace(ch = getchar()))
```

Listing 17.17 : Utilisation des macros isxxx() pour implémenter une fonction de lecture au clavier d'un nombre entier *(suite)*

```
           ;

       /* Si le premier caractère n'est pas numérique,
          le recracher et renvoyer 0 */

       if (ch != '-' && ch != '+' && !isdigit(ch) && ch != EOF)
       {
           ungetc(ch, stdin);
           return 0;
       }

       /* Si le premier caractère est un signe moins, placer
          le signe du résultat. */

       if (ch == '-')
           sign = -1;

       /* Si le premier caractère est un signe + ou un signe -,
          lire le caractère suivant */

       if (ch == '+' || ch == '-')
           ch = getchar();

       /* Lire des caractères jusqu'à en trouver un qui ne soit
          pas un chiffre. Effectuer la conversion en multipliant
          chacun des chiffres lus par la bonne puissance de 10 */
       for (i = 0; isdigit(ch); ch = getchar())
           i = 10 * i +  (ch - '0');

       /* Corriger éventuellement le signe. */

       i *= sign;

       /* Si on n'a pas rencontré d'EOF, c'est qu'on a lu un
          caractère non numérique. Le recracher. */

       if (ch != EOF)
           ungetc(ch, stdin);

       /* Renvoyer la valeur finale. */

       return i;
   }
```

```
-123
Vous avez tapé : -123.
BABA57
Vous avez tapé : 0.
685
Vous avez tapé : 685.
9 9 9
Vous avez tapé : 9.
```

Analyse

Aux lignes 31 et 61, ce programme utilise la fonction `ungetc()` que nous avons étudiée au Chapitre 14, pour "recracher" un caractère qui ne lui convient pas dans le flot d'entrée, ici `stdin`. Ce sera le premier lu dans l'ordre de lecture suivant.

Ici, c'est la fonction `get int()` qui est la plus élaborée, `main()` ne faisant que lui servir de faire-valoir. La ligne 23 boucle sur un `while` pour ignorer les espaces placés éventuellement en tête. À la ligne 29, on vérifie que le caractère lu est un de ceux qui conviennent pour représenter un entier. Si ce n'est pas le cas, il est recraché à la ligne 31 et on revient au programme appelant.

Le signe est traité aux lignes 38 à 45. La variable `sign` a été définie comme un `int` initialisé à 1. Si on trouve un signe ( ), l'instruction située à la ligne 39 lui donne la valeur –1. Il servira à multiplier la valeur absolue du nombre, en fin de conversion. Si on a lu un signe, il faut continuer à lire, en espérant trouver au moins un chiffre (lignes 44 et 45).

Le cœur de la fonction est constitué par la boucle `for` des lignes 50 et 51, qui continue à lire des caractères (`ch = getchar()`) tant que ceux-ci sont utilisables [condition terminale : `isdigit(ch)`]. La conversion s'effectue à la ligne 51 par l'instruction :

```
i = 10 * i +  (ch - '0');
```

dans laquelle la conversion du caractère en chiffre décimal s'effectue en soustrayant la valeur "caractère" 0 de `ch`. Nous n'aurions pas pu écrire :

```
i = 10 * i + strtol(ch, NULL, 10);
```

puisque `ch` est un caractère et non une chaîne de caractères (il n'est pas suivi par un terminateur `\0`).

C'est à la ligne 55 qu'on tient compte du signe et le résultat final est renvoyé à la ligne 65, le programme ayant fait un peu de nettoyage entre-temps (si le dernier caractère lu n'était pas l'équivalent d'un EOF, on le recrache).

Si compliquée soit-elle, cette fonction "oublie" de s'assurer que le nombre final obtenu est bien représentable dans un `int`, c'est-à-dire qu'il n'est pas trop grand.



À ne pas faire

Utiliser des fonctions non conformes à une norme répandue telle que ANSI, POSIX ou BSD si vous envisagez de porter vos programmes sur d'autres plates-formes.

Confondre les caractères avec les nombres : '1' est différent de 1.

Résumé

Dans ce chapitre, vous avez pu découvrir de nombreuses façons de manipuler des chaînes de caractères. À l'aide des fonctions de la bibliothèque standard vous pouvez copier, concaténer et comparer des chaînes de caractères. Vous pouvez aussi y rechercher la présence de caractères ou de groupes de caractères. Ce sont là des tâches qu'on rencontre dans beaucoup de programmes. La bibliothèque standard contient aussi des fonctions de conversion de casse (minuscule en majuscule et inversement) et de conversion de chaînes de caractères en valeurs numériques. Enfin, il existe des fonctions (plus exactement, des macros) de test de caractères.

Q & R

Q Comment puis-je savoir qu'une fonction est reconnue par tel ou tel standard ?

R La plupart des compilateurs sont fournis avec un manuel de référence de leur bibliothèque dans lequel vous trouverez la liste des fonctions avec leur description complète. Sur Linux et certains Unix, regardez le volet `Conformité` des pages de manuel.

Q Y a-t-il d'autres fonctions de traitement de caractères qui n'ont pas été présentées dans ce chapitre ?

R Oui, mais celles que nous avons vues couvrent virtuellement tous vos besoins. Consultez le manuel de votre compilateur pour connaître celles qui existent en plus.

Q Est-ce que `strcat()` ignore les espaces éventuels à droite en concaténant deux chaînes ?

R Non. Pour elle, un espace est un caractère comme un autre.

Q Puis-je convertir une valeur numérique en chaîne de caractères ?

R Bien sûr, avec la fonction `sprintf()`.

Atelier

L'atelier vous propose de tester vos connaissances sur les sujets abordés dans ce chapitre.

Quiz

1. Qu'est-ce que la longueur d'une chaîne et comment peut-on en connaître la valeur ?
2. Avant de copier une chaîne de caractères, de quoi devez-vous vous assurer ?
3. Que signifie le terme "concaténer" ?

4. Lorsque vous comparez des chaînes de caractères, que signifie "Une des deux chaînes est plus grande que l'autre" ?
5. Quelle différence y a-t-il entre `strcmp()` et `strncmp()` ?
6. Quelle différence y a-t-il entre `strcmp()` et `strcmpi()` ?
7. Quelles valeurs la fonction `isascii()` teste-t-elle ?
8. D'après le Tableau 17.4, quelles sont les macros qui renvoient vrai pour `var` si cette variable est définie par :

```
int var = 1;
```

9. D'après le Tableau 17.4, quelles sont les macros qui renvoient vrai pour `x` si cette variable est définie par :

```
char x = 65;
```

10. À quoi servent les fonctions de test de caractères ?

Exercices

1. Quelles valeurs renvoient les fonctions de test ?
2. Que va renvoyer la fonction `strtol(valeur, NULL, 10)` si on lui passe les valeurs suivantes :
   a) "65".
   b) "81.23".
   c) "–34.2".
   d) "dix".
   e) "+12mille".
   f) "moins100".
3. Que va renvoyer la fonction `strtod(valeur, NULL)` si on lui passe les valeurs suivantes :
   a) "65".
   b) "81.23".
   c) "–34.2".
   d) "dix".
   e) "+12mille".
   f) "1e+3".

4. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions qui suivent ?

```
char *string1, string2;
string1 = "Hello, World";
strcpy(string2, string1);
printf("%s %S", string1, string2);
```

Pour les exercices qui suivent, il y a plusieurs solutions possibles. Nous n'en donnerons pas le corrigé.

5. Écrivez un programme qui demande à l'utilisateur son nom, son premier prénom et son second prénom (s'il en a un). Placez-les ensuite tous trois dans une chaîne de caractères sous la forme : initiale du prénom, point, espace, initiale du second prénom, point, espace et nom. Par exemple, si l'utilisateur a tapé "Jules Oscar Hamel", vous devez obtenir "J. O. Hamel". Affichez le résultat.
6. Écrivez un programme qui prouve vos réponses aux questions 8 et 9 du quiz.
7. La fonction `strstr()` trouve la première occurrence d'une chaîne à l'intérieur d'une autre chaîne et différencie les minuscules des majuscules. Écrivez une fonction qui fasse la même chose sans distinguer les minuscules des majuscules.
8. Écrivez une fonction qui compte le nombre d'occurrences d'une chaîne à l'intérieur d'une autre chaîne.
9. Écrivez un programme qui cherche dans un fichier texte les occurrences d'une chaîne de caractères spécifiée par l'utilisateur, et affiche les numéros des lignes répondant au critère. Par exemple, si vous recherchez dans un de vos programmes C les occurrences de la chaîne `printf`, votre programme devra afficher toutes les lignes faisant appel à la fonction `printf()`. Sur Unix, contrôlez le résultat avec la commande `grep` qui effectue la même chose.
10. Écrivez une fonction `get float()` analogue à celle du Listing 17.17, mais donnant le résultat sous forme de nombre flottant.

18

Retour sur les fonctions

Comme vous avez dû vous en apercevoir, les fonctions constituent l'un des éléments les plus importants de la puissance et de la souplesse du langage C. Dans ce chapitre, nous allons voir les points suivants :

- Utilisation de pointeurs comme arguments de fonction
- Passage de pointeurs de type `void` en arguments
- Utilisation de fonctions avec un nombre variable d'arguments
- Renvoi d'un pointeur par une fonction

Nous avons déjà étudié certains de ces sujets dans les chapitres précédents, mais nous allons y revenir plus en détail dans ce chapitre.

Passage de pointeurs à une fonction

La méthode par défaut pour passer un argument à une fonction est un passage *par valeur*, ce qui signifie que la fonction reçoit une copie de la valeur de l'argument. Cette méthode se réalise en trois étapes :

1. L'expression à passer en argument est évaluée (ou prise telle quelle s'il s'agit d'une constante ou d'une variable).
2. Le résultat est recopié sur la pile (*stack*) qui est une zone de stockage temporaire en mémoire.
3. La fonction récupère la valeur de l'argument sur la pile.

Cette procédure est illustrée par la Figure 18.1 avec un seul argument : une variable de type `int` ; mais la méthode est la même pour d'autres types de variables et des expressions plus complexes.

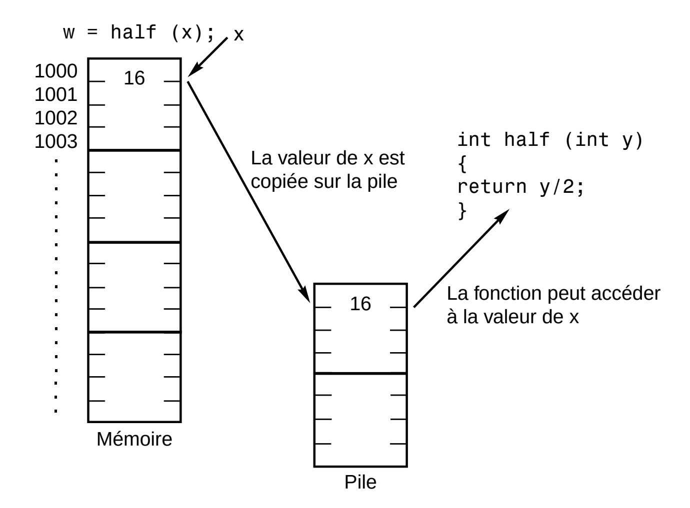


Figure 18.1
Passage d'un argument par valeur. La fonction ne peut pas modifier la valeur originale de la variable.

Lorsqu'une variable est passée à une fonction par valeur, la fonction a accès à la valeur de la variable, mais pas à la variable elle-même. Il en résulte que la fonction ne peut pas modifier la valeur originale de cette variable. C'est la principale raison d'utiliser un passage d'argument *par valeur*. Les données situées à l'extérieur d'une fonction sont protégées de toute altération accidentelle.

Ce mode de transmission d'arguments est possible avec les types de données de base (`char`, `long`, `float` et `double`) et les structures. Il existe, cependant, une méthode pour passer des arguments à une fonction, consistant à passer un *pointeur* vers la variable

originale et non vers une copie de sa valeur. Cette méthode est appelée *passage par adresse* (ou par *référence*).

Comme nous l'avons vu au Chapitre 9, ce type de passage est le seul utilisable pour les tableaux. Les variables simples et les structures peuvent être transmises indifféremment par les deux méthodes. Si votre programme utilise de grosses structures, les passer par valeur risquerait d'entraîner un débordement de la pile. Passer un argument par adresse permet à la fonction de modifier la valeur originale de la variable, ce qui est à la fois un avantage et un inconvénient.

Que ce soit à la fois un avantage et un inconvénient ne doit pas vous étonner. Tout dépend de la situation dans laquelle on se trouve. Si le programme a besoin de modifier la valeur de la variable, alors, c'est un avantage. Si ce n'est pas le cas, cela peut dégénérer en inconvénient.

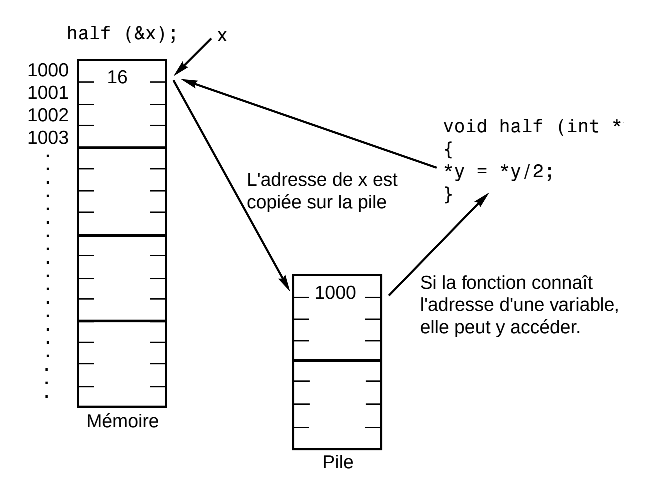
Il existe une façon bien simple de modifier la valeur d'un argument, comme le montrent ces quelques lignes de programme :

```
x = f(x)

int f(int y)
{ return y/2;
}
```

Souvenez-vous qu'une fonction ne peut renvoyer qu'une seule valeur. En passant un ou plusieurs arguments par adresse, permettez à une fonction de renvoyer plusieurs résultats au programme qui l 'a appelée. La Figure 18.2 illustre le passage par adresse d'un seul argument.

Figure 18.2
Le passage par adresse permet à la fonction de modifier la valeur originale de la variable.



La fonction utilisée pour la Figure 18.2 ne donne pas un bon exemple d'utilisation d'un appel par adresse dans un véritable programme, mais elle illustre bien le concept. Lorsque vous passez un argument par adresse, il faut vous assurer que le prototype de la fonction indique bien cette méthode de passage (on passe non plus une valeur, mais un pointeur). Dans le corps de la fonction, vous devrez aussi utiliser l'opérateur d'indirection pour accéder à la variable (ou aux variables) passée(s) par adresse.

Le programme du Listing 18.1 montre un passage d'arguments par adresse et par valeur, et prouve clairement que cette dernière méthode ne permet pas d'altérer la valeur initiale de la variable. Bien entendu, une fonction n'est pas obligée de modifier la valeur d'un argument transmis par adresse, mais "elle peut le faire".

Listing 18.1 : Passage par valeur et passage par adresse

```
 /* Passage d'arguments par valeur et par adresse.
 */
 #include <stdio.h>
 #include <stdlib.h>
 void par_valeur(int a, int b, int c);
 void par_adresse(int *a, int *b, int *c);

 int main()
 {
    int x = 2, y = 4, z = 6;

    printf("\nAvant d'appeler par_valeur (), x = %d, y = %d,\
z = %d.", x, y, z);

    par_valeur(x, y, z);

    printf("\nAprès avoir appelé par_valeur(), x = %d, y = %d,
z = %d.", x, y, z);

    par_adresse(&x, &y, &z);
    printf("\nAprès avoir appelé par_adresse(), x = %d, y = %d,
z = %d.\n", x, y, z);
    exit(EXIT_SUCCESS);
}
void par_valeur(int a, int b, int c)
{
    a = 0;
    b = 0;
    c = 0;
}

```

```
33:   void par_adresse(int *a, int *b, int *c)
34:   {
35:       *a = 0;
36:       *b = 0;
37:       *c = 0;
38:   }
```



```
Avant d'appeler par_valeur (), x = 2, y = 4, z = 6.
Après avoir appelé par_valeur(), x = 2, y = 4, z = 6.
Après avoir appelé par_adresse(), x = 0, y = 0, z = 0.
```

Analyse

Ce programme montre la différence entre les deux méthodes de passage d'arguments. Les lignes 5 et 6 contiennent les prototypes des deux fonctions qu'appellera le programme. La première comporte une liste d'arguments "ordinaires", de type `int` : c'est l'appel "par valeur" (comme l'indique le nom de la fonction). La liste des arguments de l'autre fonction est une liste de pointeurs. Il s'agit d'appels "par adresse". Aux lignes 26 et 33, la première instruction de chacune des fonctions respecte la déclaration faite par le prototype. On fait la même chose dans le corps de ces fonctions, mais on ne le fait pas de la même façon. Les deux fonctions remettent à 0 les variables transmises. Dans la première, il s'agit des copies, placées sur le stack, des valeurs originales ; dans la seconde, on remet à zéro les variables elles-mêmes, dans le programme appelant.

À la ligne 12, on affiche les valeurs originales des variables. Ensuite, à la ligne 15, on appelle la fonction `par valeur()` et on réaffiche les valeurs des variables à la ligne 17. On remarque qu'elles n'ont pas changé. À la ligne 20, on appelle la fonction `par adresse()` puis, à la ligne 22, on réaffiche les valeurs. On constate alors qu'elles valent bien zéro. La preuve est faite qu'on peut modifier les valeurs des variables du programme appelant.

Il est possible de mélanger les deux types de passage d'arguments dans un même appel de fonction. Il faudra bien faire attention, en appelant plus tard la fonction, à ne pas les mélanger.



À faire

Passer des variables par valeur pour ne pas risquer de les voir modifiées par la fonction appelée.

À ne pas faire

Passer de grandes quantités de variables par valeur lorsque ce n'est pas nécessaire. Vous risqueriez de faire déborder la pile.

Oublier qu'une variable passée par adresse doit être écrite sous forme de pointeur. Dans la fonction, vous devrez utiliser l'opérateur d'indirection pour récupérer sa valeur.

Les pointeurs de type *void*

Vous avez déjà rencontré le mot clé `void` servant à indiquer qu'une fonction ne renvoie pas de valeur ou ne demande pas d'argument. On peut aussi l'utiliser pour déclarer un pointeur de type générique, c'est-à-dire pouvant pointer vers n'importe quel type d'objet. Par exemple, l'instruction :

```
void *x;
```

déclare un pointeur générique `x`, pointant vers un objet dont on n'a pas encore spécifié le type.

L'usage le plus fréquent de ce type de pointeurs se rencontre dans une déclaration des arguments d'une fonction. Vous pouvez souhaiter créer une fonction capable de prendre en compte différents types d'arguments : `int`, `float`, etc. En déclarant le pointeur vers cet argument comme étant de type `void`, vous pouvez utiliser plusieurs types d'arguments.

Voici un exemple simple. Vous voulez écrire une fonction, `demi()` calculant la moitié de la valeur qui lui est passée, et vous voulez que son résultat soit du même type. Outre la valeur elle-même, vous lui passez un caractère indiquant le type de l'argument ('i' pour `int`, 'f' pour `float`, etc.). La fonction sera alors déclarée de la façon suivante :

```
void demi(void *x, char type);
```

Mais, avoir passé un pointeur est une chose, pouvoir récupérer correctement la valeur vers laquelle il pointe en est une autre. Il faut, pour cela, *caster* ce pointeur afin que le compilateur sache à quoi s'en tenir et fasse son travail correctement. Dans le cas d'un entier, par exemple, on écrira :

```
(int *) x
```

et pour accéder à la *valeur* de la variable, il faut rajouter l'opérateur d'indirection. Cela donne :

```
* (int *) x
```

Le casting sera approfondi au Chapitre 20.

Pour notre exemple, nous avons retenu quatre types de variables : `int`, `long`, `float` et `double`, indiqués respectivement par leur initiale sous forme de constante caractère.

Le Listing 18.2 vous présente la fonction `demi()` accompagnée d'un `main()` simple pour la tester.

Listing 18.2 : Utilisation d'un pointeur de type void pour passer des variables de types différents à une fonction

```
 /* Utilisation de pointeurs de type void. */
 #include <stdio.h>
 #include <stdlib.h>

 void demi(void *x, char type);

 int main()
 {
    /* Initialiser une variable de chaque type. */

    int i = 20;
    long l = 100000L;
    float f = 12.456;
    double d = 123.044444;

    /* Afficher leur valeur initiale. */

    printf("\n%d", i);
    printf("\n%ld", l);
    printf("\n%f", f);
    printf("\n%lf\n\n", d);

    /* Appeler demi() pour chaque variable. */

    demi(&i, 'i');
    demi(&l, 'l');
    demi(&d, 'd');
    demi(&f, 'f');

    /* Afficher leur nouvelle valeur. */
    printf("\n%d", i);
    printf("\n%ld", l);
    printf("\n%f", f);
    printf("\n%lf", d);
    exit(EXIT_SUCCESS);
}

void demi(void *x, char type)
{
    /* Caster la valeur de x selon le type de x et faire
       la division par 2 */

    switch (type)
    {   case 'i':
            *(int *) x /= 2;
            break;
        case 'l':
            *(double *) x /= 2L;
```

Listing 18.2 : Utilisation d'un pointeur de type void pour passer des variables de types différents à une fonction *(suite)*

```
            break;
        case 'f':
            *(float *) x /= 2.;
            break;
        case 'd':
            *(double *) x /= 2.0;
            break;
    }
}
```

```
20
100000
12.456000
123.044444


5
50000
6.228000
61.522222
```

Analyse

Pour simplifier, on ne fait aucun test de validité, ni sur les arguments, ni sur l'indicateur de type. Il aurait été prudent de rajouter une clause `default` au `switch` :

```
default :
         printf("%c : type inconnu.\n", type);
```

(Rappelons qu'il est inutile de terminer cette clause par un `break;` puisque c'est la dernière du `switch`.)

Vous pourriez penser que la nécessité de passer le type de variable en plus de sa valeur diminue la souplesse de cette fonction, et qu'elle aurait été plus générale sans cette indication complémentaire. Malheureusement, C ne dispose pas d'une boule de cristal lui permettant de deviner le type de la variable qui lui est passée. Dans le mécanisme de transmission des arguments, il n'a pas été prévu d'indicateur de type. Il faut donc y suppléer par des astuces comme celle que nous venons de voir.

On pourrait réécrire cette fonction sous forme de macro. Nous le ferons au Chapitre 21.

Comme on ne connaît pas la *longueur* de la variable sur laquelle pointe le pointeur de type `void`, on ne peut ni l'incrémenter, ni le décrémenter. (En effet, si `p` est un pointeur de type `void`, `sizeof(*p)` n'a pas de sens et ne peut servir qu'à fâcher le compilateur.)

Conseils

À faire

Penser à caster le pointeur de type `void` *pour utiliser la valeur sur laquelle il pointe.*

Maîtriser ses arguments : il est rare d'avoir à indiquer le type des arguments. En l'occurrence, de toutes les fonctions standard que nous avons vues, seules celles de la famille de `printf()` *utilisent ce principe.*

À ne pas faire

Tenter d'incrémenter ou de décrémenter un pointeur de type `void`*.*

Utiliser des pointeurs de type `void` *à tort et à travers.*

Fonctions avec un nombre variable d'arguments

Vous avez souvent utilisé des fonctions de bibliothèque (`printf()` et `scanf()`, par exemple) dont le nombre d'arguments est variable. Vous pouvez écrire de telles fonctions vous-même. Il faudra alors inclure le fichier d 'en-tête stdarg.h.

Pour déclarer une fonction admettant un nombre variable d'arguments, commencez par faire apparaître les arguments fixes : ceux qui doivent toujours figurer (il doit y en avoir au moins un). Ensuite, écrivez une *ellipse*, c'est-à-dire trois points à la suite, pour indiquer la présence d'un nombre variable d'arguments, éventuellement nul.

Il faut que la fonction ait un moyen de savoir combien d'arguments lui ont été passés à chaque appel. On pourrait songer à inclure, parmi les arguments de la fonction, une variable qui spécifierait ce nombre. Outre l'inélégance de ce procédé, cela ne mettrait pas l'utilisateur à l'abri d'une erreur dans le comptage de ses arguments, et ôterait beaucoup de sûreté et de rigueur aux programmes qu'il pourrait écrire.

On pourrait aussi, comme dans `printf()`, tabler sur les spécificateurs de conversion (% quelque chose) pour connaître le nombre des variables qui suivent le format. Mais, on retrouve le même inconvénient que précédemment. Et cela ne permettrait pas la reconnaissance du type des variables transmises. Bien sûr, avec la "méthode" `printf`, on pourrait le déduire de l'indicateur de conversion. Mais, tout cela reste du bricolage et il s'agit là d'un cas très particulier à partir duquel on ne peut pas généraliser.

Heureusement, le C propose des outils pour réaliser, avec élégance et sécurité, ce passage d'arguments en nombre variable. Ces outils se trouvent dans stdarg.h et sont les suivants :

`va list`	Type de pointeur vers des données.
`va start()`	Macro servant à initialiser la liste des arguments.

`va arg()`	Macro servant à récupérer chaque argument, tour à tour, de la liste des variables.
`va end()`	Macro servant à "faire du nettoyage", une fois tous les arguments récupérés.

Pour bien utiliser ces outils, il faut respecter attentivement les étapes suivantes, illustrées par le programme du Listing 18.3 :

1. Déclarez un pointeur de type `va list`. Il servira à accéder aux arguments individuels. Bien que ce ne soit pas obligatoire, il est d'usage de l'appeler `arg ptr`.
2. Appelez la macro `va start()` en lui passant `va list` ainsi que le nom du dernier argument fixe. Cette macro ne renvoie pas de valeur : elle se contente d'initialiser le pointeur `arg ptr`, qui va ensuite pointer vers le premier argument de la liste variable.
3. Pour récupérer, tour à tour, chacun des arguments, appelez `va arg()` en lui passant le pointeur `arg ptr` et le type de données de l'argument suivant (`int`, `long`, `double`...). La valeur de retour de `va arg()` est la valeur de l'argument suivant. Si la fonction a été appelée avec `n` arguments dans la liste variable, vous devrez appeler `n` fois `va arg()` et vous récupérerez, dans l'ordre où ils ont été écrits, les arguments de la liste variable.
4. Quand vous avez récupéré tous les arguments de la liste variable, il faut "fermer la liste" et, pour cela, appeler `va end()` en lui passant le pointeur `arg ptr`. Dans certaines implémentations, cette macro ne fait rien ; dans d'autres, elle déblaie le terrain. Quoi qu'il en soit, il faut systématiquement l'appeler sous peine d'aboutir à la réalisation de programmes non portables.

La fonction `moyenne()` du Listing 18.3 calcule la moyenne arithmétique d'une suite de nombres entiers. Ce programme transmet à la fonction un argument fixe, en indiquant le nombre d'arguments supplémentaires suivi de la liste des nombres.

Listing 18.3 : Utilisation d'une liste d'arguments variables

```
/* Fonctions avec un nombre variable d'arguments. */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

float moyenne(int num, ...);

int main()
{
   float x;

   x = moyenne(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
13:         printf("\nLa première moyenne est %f.", x);
14:         x = moyenne(5, 121, 206, 76, 31, 5);
15:         printf("\nLa seconde moyenne est %f.\n", x);
16:         exit(EXIT_SUCCESS);
17:     }
18:
19:     float moyenne(int num, ...)
20:     {
21:         /* Déclarer une variable de type va_list. */
22:
23:         va_list arg_ptr;
24:         int count, total = 0;
25:
26:         /* Initialiser le pointeur vers les arguments. */
27:
28:         va_start(arg_ptr, num);
29:
30:         /* Récupérer chaque argument de la liste variable. */
31:
32:         for (count = 0; count < num; count++)
33:             total += va_arg(arg_ptr, int);
34:
35:         /* Donner un coup de balai. */
36:
37:         va_end(arg_ptr);
38:
39:         /* Diviser le total par le nombre de valeurs à moyenner.
40:            Caster le résultat en float puisque c'est le type
41:            du résultat.  */
42:
43:         return ((float)total/num);
44:     }
```

Voici le résultat obtenu :

```
La première moyenne est 5.500000.
La seconde moyenne est 87.800000.
```

Analyse

La fonction `moyenne()` est appelée une première fois à la ligne 12. Le premier argument passé, le seul qui soit constant, spécifie le nombre de valeurs de la liste variable. C'est aux lignes 32 et 33 que vont avoir lieu la récupération et la sommation de chacun des arguments. Une fois tous les arguments récupérés, la ligne 43 opère une division par le nombre d'éléments, et caste le résultat en `float` avant de le renvoyer au programme appelant.

À la ligne 28, on appelle `va start()` pour initialiser le mécanisme de récupération. Cet appel doit précéder les appels à `va arg()`. Enfin, à la ligne 37, on referme par un appel à `va end()` qui fera éventuellement un peu de nettoyage.

En réalité, une fonction n'a pas nécessairement besoin de placer le nombre d'arguments dans la liste de ses arguments. On pourrait imaginer d'autres systèmes de délimitations : une dernière valeur négative ou nulle, lorsqu'aucun des éléments de la liste ne risque de l'être. Mais faire figurer en tête de liste le nombre d'arguments est une méthode plus générale ; c'est celle qui doit normalement être employée.

Fonctions renvoyant un pointeur

Vous avez déjà rencontré, dans la bibliothèque standard, des fonctions renvoyant un pointeur au programme qui les a appelées. Rien ne vous empêche d'écrire de telles fonctions. Comme vous pouvez vous y attendre, l'opérateur d'indirection (`*`) est utilisé à la fois dans la déclaration de la fonction et dans sa définition. Voici quelle en est la forme générale :

```
type *fonc(liste d'arguments);
```

Cette instruction déclare une fonction `fonc()` qui renvoie un pointeur vers un `type`. En voici deux exemples concrets :

```
double *fonc1(liste d'arguments);
struct adresse *fonc2(liste d'arguments);
```

La première ligne déclare une fonction qui renvoie un pointeur vers un type `double`. La seconde ligne déclare une fonction qui renvoie un pointeur vers un type `adresse` (supposé défini par l'utilisateur).

Ne confondez pas *une fonction qui renvoie un pointeur* avec *un pointeur vers une fonction*. Si vous mettez une paire de parenthèses supplémentaire dans la déclaration, c'est la dernière forme que vous déclarez, comme on le voit dans ces deux exemples :

```
double (*fonc)(...); // pointeur vers une fonction qui renvoie un double
double *fonc(...);  // fonction qui renvoie un pointeur vers un double
```

Maintenant que nous savons déclarer correctement notre fonction, une question se pose. Comment allons-nous l'utiliser ? Il n'y a rien de spécial à préciser au sujet de ces fonctions : on les utilise comme n'importe quelle autre fonction, en assignant leur valeur de retour à une variable du type approprié (donc un pointeur). Comme, en C, un appel de fonction est une expression, vous pouvez vous en servir à n'importe quel endroit de votre programme.

Le Listing 18.4 présente un exemple simple. Il s'agit d'une fonction à laquelle on passe deux arguments et qui détermine et renvoie le plus grand. On verra qu'il y a en réalité deux exemples : l'un retourne un `int` ; l'autre, un pointeur vers un `int`.

Listing 18.4 : Renvoi d’un pointeur par une fonction

```
/* Fonction qui retourne un pointeur. */
#include <stdio.h>
#include <stdlib.h>

int superieur1(int x, int y);
int *superieur2(int *x, int *y);

int main()
{
   int a, b, plusgrand1, *plusgrand2;

   printf("Tapez deux nombres entiers : ");
   scanf("%d %d", &a, &b);

   plusgrand1 = superieur1(a, b);
   printf("\nCelui qui a la plus grande valeur est : %d.",
           plusgrand1);
   plusgrand2 = superieur2(&a, &b);
   printf("\nCelui qui a la plus grande valeur est : %d.\n",
           *plusgrand2);
   exit(EXIT_SUCCESS);
}
int superieur1(int x, int y)
{
   if (y > x)
       return y;
   return x;
}

int *superieur2(int *x, int *y)
{
   if (*y > *x)
       return y;

   return x;
}
```

```
Tapez deux nombres entiers : 1111 3000
Celui qui a la plus grande valeur est : 3000.
Celui qui a la plus grande valeur est : 3000.
```

Analyse

La logique de ce programme n’exige pas de grandes explications. On s’attardera un peu sur l’écriture des deux fonctions : `superieur1()` et `superieur2()`. Si celle de la première est d’un type "classique", pour la seconde, on voit qu’on a fait usage de pointeurs à la fois pour les arguments et pour le résultat.

Bien entendu, l'appel de `superieur2()` doit être fait avec des valeurs transmises par adresse (ligne 18) et l'affichage du résultat (ligne 19) nécessite l'emploi de l'opérateur d'indirection (`*`).



À faire

Utiliser les éléments que nous venons d'étudier pour écrire des fonctions avec un nombre variable d'arguments, même si le compilateur n'exige pas tous ces éléments.

Encadrer les appels à `va arg()` par un appel initial à `va start()` et un appel final à `va end()`.

À ne pas faire

Confondre un pointeur vers une fonction avec une fonction qui renvoie un pointeur.

Résumé

Dans ce chapitre, vous avez étudié quelques particularités concernant les fonctions : la différence entre passer des arguments par valeur et les passer par adresse. Vous avez vu comment cette dernière technique permettait à une fonction de renvoyer *plusieurs* résultats. Vous avez appris à utiliser le type `void` pour créer un pointeur générique capable de pointer sur n'importe quel objet de données du langage C. Ce type de pointeur est le plus souvent utilisé avec des fonctions auxquelles on peut passer des arguments de types différents. Rappelez-vous qu'il est nécessaire de caster un pointeur de type `void` avant de pouvoir utiliser l'objet sur lequel il pointe.

Vous avez découvert les macros contenues dans stdarg.h et vous savez maintenant comment les utiliser pour créer des fonctions admettant un nombre variable d'arguments. Ces fonctions apportent beaucoup de souplesse à la programmation. Enfin, nous avons vu ce qu'était une fonction renvoyant un pointeur et comment l'utiliser.

Q & R

Q Est-il courant, lorsqu'on programme en C, de passer des pointeurs ?

R Absolument ! Dans beaucoup de cas, une fonction a besoin de modifier plusieurs variables. Il y a deux façons d'y parvenir. La première consiste à déclarer et à utiliser des variables globales. La seconde, à passer des pointeurs de façon que la fonction puisse

directement modifier les variables ; la première option est surtout intéressante si plusieurs fonctions utilisent les mêmes variables (voir Chapitre 12).

Q Vaut-il mieux modifier une valeur en la renvoyant ou en passant un pointeur sur elle ?

R Quand il n'y a qu'une valeur à modifier dans une fonction, il est d'usage de se servir pour cela de la valeur de retour. En revanche, si vous devez changer la valeur de plusieurs variables, soit vous utilisez les structures en renvoyant un pointeur sur une, soit vous passez un pointeur sur ces variables.

Atelier

L'atelier vous propose quelques questions permettant de tester vos connaissances sur les sujets que nous venons d'aborder dans ce chapitre.

Quiz

1. Lorsque vous passez des arguments à une fonction, quelle est la différence entre les passer par valeur et les passer par adresse ?
2. Qu'est-ce qu'un pointeur de type `void` ?
3. Pour quelle raison utilise-t-on un pointeur de type `void` ?
4. Lorsque vous utilisez un pointeur de type `void`, pour quelle raison emploie-t-on un casting et quand doit-on le faire ?
5. Pouvez-vous écrire une fonction ne contenant qu'une liste variable d'arguments, sans argument défini une fois pour toutes ?
6. Quelles macros devez-vous utiliser lorsque vous écrivez des fonctions ayant une liste d'arguments variable ?
7. Quelle est la valeur ajoutée à un pointeur de type `void` lorsqu'il est incrémenté ?
8. Est-ce qu'une fonction peut renvoyer un pointeur ?

Exercices

1. Écrivez le prototype d'une fonction qui renvoie un entier. Elle aura comme argument un pointeur vers un tableau de caractères.
2. Écrivez un prototype pour une fonction appelée `nombres()` qui accepte trois entiers comme arguments, lesquels seront passés par adresse.

3. Comment écririez-vous l'instruction d'appel de la fonction `nombres()` de l'exercice précédent avec trois entiers : `ent1`, `ent2` et `ent3` ?
4. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions qui suivent ?

```
void carre(int *nombre, ...)
{ *nombre *= *nombre;
}
```

5. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions qui suivent ?

```
float total(int nombre, ...)
{ int compte, total=0;

  for (compte=0; compte<nombre; compte++)
     total += va_arg(arg_ptr, int);
  return total;
}
```

Les exercices suivants admettent plusieurs solutions. Nous n'en donnerons pas les corrigés.

6. Écrivez une fonction à laquelle on passe un nombre variable de chaînes de caractères en argument, qui concatène ces chaînes, dans l'ordre, en une seule et unique chaîne et renvoie un pointeur vers la nouvelle chaîne.
7. Écrivez une fonction à laquelle on passe un tableau de nombres de n'importe quel type en argument, qui recherche le plus grand et le plus petit des nombres du tableau et renvoie des pointeurs vers ces valeurs. (Aide : Vous devrez trouver un moyen d'indiquer à cette fonction combien il y a d'éléments dans le tableau.)
8. Écrivez une fonction qui accepte une chaîne de caractères et un caractère isolé. Elle recherche la première occurrence du caractère dans la chaîne et renvoie un pointeur vers cet emplacement.

19

Exploration de la bibliothèque des fonctions

Comme vous l'avez vu tout au long de ce livre, une grande partie de la puissance de C vient de la bibliothèque standard des fonctions. Dans ce chapitre, nous allons étudier des fonctions qui n'entrent pas dans les sujets abordés par les autres chapitres :

- Les fonctions mathématiques
- Les fonctions qui concernent le temps
- Les fonctions de traitement d'erreur
- Les fonctions de recherche de données et de tri

Les fonctions mathématiques

La bibliothèque standard contient un certain nombre de fonctions destinées à réaliser des opérations mathématiques. Leurs prototypes se trouvent dans le fichier d 'en-tête math.h. Toutes renvoient un résultat de type `double`. En ce qui concerne les fonctions trigonométriques, elles portent sur des angles exprimés en radians. Souvenez-vous qu'un radian vaut 180˚ / 3.14159265, soit 57, 296˚.

Fonctions trigonométriques

Fonction	*Prototype*	*Description*
`acos()`	`double acos(double x)`	Renvoie l'arc cosinus d'un argument `x` tel que –1 `<= x <= 1`
`asin()`	`double asin(double x)`	Renvoie l'arc sinus d'un argument `x` tel que –1 `<= x <= 1`
`atan()`	`double atan(double x)`	Renvoie l'arc tangente de l'argument `x`
`atan2()`	`double atan2 (double x, double y)`	Renvoie l'arc tangente du rapport `x/y`
`cos()`	`double cos(double x)`	Renvoie le cosinus de l'argument `x`
`sin()`	`double sin(double x)`	Renvoie le sinus de l'argument `x`
`tan()`	`double tan(double x)`	Renvoie la tangente de l'argument `x`

Fonctions logarithmiques et exponentielles

Fonction	*Prototype*	*Description*
`exp()`	`double exp(double x)`	Renvoie e^x (e = 2,71828)
`log()`	`double log(double x)`	Renvoie le logarithme naturel de x
`log10()`	`double log10(double x)`	Renvoie le logarithme vulgaire (base 10) de x
`frexp ()`	`double frexp(double x, int *y)`	Décompose x en une fraction normalisée (représentée par la valeur de retour) et une puissance entière de 2, y. Si x = 0, les deux parties du résultat valent 0
`ldexp()`	`double ldexp(double x, int y)`	Renvoie x multiplié par 2y

Fonctions hyperboliques

Fonction	*Prototype*	*Description*
`cosh()`	`double cosh(double x)`	Renvoie le cosinus hyperbolique de l'argument x
`sinh()`	`double sinh(double x)`	Renvoie le sinus hyperbolique de l'argument x
`tanh()`	`double tanh(double x)`	Renvoie la tangente hyperbolique de l'argument x

Autres fonctions mathématiques

Fonction	*Prototype*	*Description*
`sprt()`	`double sqrt(double x)`	Renvoie la racine carrée de l'argument `x` qui doit être positif ou nul
`ceil()`	`double ceil(double x)`	Renvoie le plus petit entier supérieur ou égal à l'argument `x`
`abs()`	`int abs(int x`	Renvoie la valeur absolue de l'argument `x`
`labs()`	`long abs(long x)`	Renvoie la valeur absolue de l'argument `x`
`floor()`	`double floor(double x)`	Renvoie le plus grand entier inférieur ou égal à l'argument `x`
`modf ()`	`double modf(double x, double *y)`	Décompose x en parties entière et décimale (du même signe que `x`). La partie entière est retournée dans l'objet pointé par `y`, et la partie décimale est la valeur de retour
`pow ()`	`double pow(double x, double y)`	Renvoie `xy`. Il y aura une erreur de domaine si `x` est négatif ou nul et si `y` n'est pas un entier ou si `x` = 0 et que `y` est négatif ou nul
`fmod ()`	`double fmod(double x, double y)`	Renvoie le reste de `x/y` avec le signe de `x`, ou 0 si `x` est nul

Exemples

Un livre entier ne suffirait pas pour illustrer toutes les utilisations possibles des fonctions mathématiques. Le Listing 19.1 ne donne que quelques exemples.

Listing 19.1 : Utilisation des fonctions mathématiques de la bibliothèque standard

```
 1:    /* Exemples d'utilisation de quelques fonctions mathématiques. */
 2:    #include <stdio.h>
 3:    #include <stdlib.h>
 4:    #include <math.h>
 5:
 6:    int main()
 7:    {
 8:
 9:        double x;
10:
11:       printf("Tapez un nombre: ");
12:       scanf("%lf", &x);
13:
14:       printf("\n\nValeur originale : %lf", x);
15:
16:       printf("\nCeil: %lf", ceil(x));
17:       printf("\nFloor: %lf", floor(x));
18:       if(x >= 0)
19:           printf("\nRacine carrée : %lf", sqrt(x));
20:       else
21:          printf("\nNombre négatif.");
22:
23:       printf("\nCosinus : %lf\n", cos(x));
24:       exit(EXIT_SUCCESS);
25:   }
```

```
Tapez un nombre:  100.95

Valeur originale : 100.950000
Ceil: 101.000000
Floor: 100.000000
Racine carrée : 10.047388
Cosinus : 0.913482
```

Analyse

La valeur entrée à la ligne 12 est affichée puis envoyée à quatre fonctions mathématiques. On notera qu'un test est effectué sur le signe de cette valeur avant d'en prendre la racine carrée.

Prenons le temps...

La bibliothèque standard contient plusieurs fonctions relatives au temps. En langage C, le temps représente aussi bien la date que l'heure. Les prototypes et la définition de la structure de ces fonctions se trouvent dans le fichier d'en-tête time.h.

Représentation du temps

Les fonctions C utilisent deux types de représentation du temps. La plus simple est le nombre de secondes écoulées depuis le 1er janvier 1970 à 0 h 00 sur le méridien de Greenwitch (soit à 1 h du matin le même jour dans le fuseau horaire de Paris). Pour les dates antérieures, la valeur est négative.

Ces valeurs sont stockées dans des variables de type `long`. Dans time.h, les symboles `time t` et `clock t` sont définis comme étant de type `long`, par un `typedef`. Ce sont eux qui sont utilisés dans les prototypes des fonctions aux lieu et place de `long`.

La seconde méthode représente le temps en le décomposant en année, mois, jour, etc. On utilise alors une structure `tm` ainsi définie :

```
struct tm
{ int   tm_sec;    // secondes [0,61]
  int   tm_min;    // minutes [0,59]
  int   tm_hour;   // heure [0,23]
  int   tm_mday;   // jour du mois [1, 31]
  int   tm_mon;    // mois [1, 12]
  int   tm_year;   // années depuis 1900
  int   tm_wday;   // jour de la semaine depuis dimanche [0, 6]
  int   tm_yday;   // jour depuis le 1er janvier [0, 165]
  int   tm_isdst;  // indicateur d'heure d'été
};
```

On ne manquera pas de s'étonner de l'intervalle [0, 61] prévu pour la valeur de `tm sec`. Il permet d'insérer un ajustement aux secondes intercalaires apportées parfois à la durée de l'année, pour compenser le ralentissement de la durée de rotation de la Terre.

Fonctions traitant du temps

Nous allons étudier les différentes fonctions de la bibliothèque standard qui traitent du temps. Ici, le mot "temps" se réfère aussi bien à la date qu'à l'heure.

Date et heure actuelles

Pour avoir le temps courant tel qu'il est maintenu par l'horloge interne de votre ordinateur, il faut appeler la fonction `time()`. Voici son prototype :

```
time_t time(time_t *timeptr);
```

Rappelez-vous que `time h` est un alias de `long`. La fonction renvoie le nombre de secondes écoulées depuis le 1er janvier 1970. Si vous lui passez un pointeur non nul, elle renseigne la structure `time t` pointée par `timeptr`. Ainsi, pour obtenir les valeurs du temps courant (`maintenant`) dans la structure de type `time t`, il suffit d'écrire :

```
time_t maintenant;
maintenant = time(0);
```

Mais vous auriez aussi pu écrire :

```
time_t maintenant;
time_t *m = &maintenant;
time(m);
```

Conversion entre les deux représentations du temps

En pratique, il n'est pas vraiment utile de connaître le nombre de secondes écoulées depuis le 1er janvier 1970, il est souvent commode de convertir cette valeur à l'aide de la fonction `localtime()`. Elle renseigne une structure de type `tm`, qu'il est ensuite facile d'afficher. Voici quel est son prototype :

```
struct tm *localtime(time_ptr *ptr);
```

Cette fonction renvoie donc un pointeur vers une structure statique de type `tm`. Il est donc inutile de déclarer une structure de ce type ; il suffit de déclarer un pointeur vers un objet de type `tm`. C'est la même structure statique qui est réutilisée à chaque appel de `local time()`, aussi, si vous voulez sauvegarder la valeur renvoyée, vous devez déclarer une structure personnelle de type `tm` dans laquelle vous recopierez le résultat de l'appel à `localtime()`.

La conversion inverse (d'une structure `tm` vers une valeur de type `time t`) s'accomplit en appelant la fonction `mktime()` dont le prototype est :

```
time_t mktime(struct tm *ntime);
```

La fonction renvoie le nombre de secondes écoulées depuis le 1er janvier 1970, et le temps représenté par la structure de type `tm` pointée par `ntime`.

Affichage du temps

Pour convertir le temps en chaînes de caractères pouvant être affichées, il existe deux fonctions : `ctime()` et `asctime()`. Les deux renvoient une chaîne de caractères d'un format spécifique. Elles diffèrent par le type d'argument qui doit leur être passé :

```
char *asctime(struct tm *ptr);
char *ctime(time_t *ptr);
```

Les deux fonctions renvoient un pointeur vers une chaîne de 26 caractères, statique, terminée par un zéro et de la forme :

```
Fri Sep 22 06:43:46 1995
```

Notez que les abréviations des jours et des mois sont en anglais (ici, par exemple, `Fri` signifie *Friday*, "vendredi"). Heureusement, l'heure est comptée "à l'européenne", entre 0 et 23 heures.

Pour mieux contrôler le format du temps, on préférera appeler la fonction `strftime()`, à laquelle on passe une structure de type `tm`, et qui formate le temps selon une chaîne de caractères spéciale. Son prototype est :

```
size_t strftime(char *s; size_t max, char *fmt, struct tm *ptr);
```

À partir du temps représenté par la structure pointée par `ptr`, la fonction formate les valeurs à afficher selon les spécifications de la chaîne pointée par `fmt` ; puis elle écrit le résultat dans une chaîne de caractères terminée par un zéro à l'emplacement mémoire pointé par `s`. Si le résultat (y compris le zéro terminal) dépasse `max` caractères, la fonction renvoie `0` et le contenu de `s` est faussé. Sinon, elle renvoie le nombre de caractères réellement écrits dans `s`, soit `strlen(s)`.

Les spécificateurs de format sont particuliers à cette fonction. Le Tableau 19.1 en donne la liste.

Tableau 19.1 : Spécificateurs de format à utiliser pour strftime()

Spécificateur	***Remplacé par***
`%a`	Nom du jour de la semaine en abrégé
`%A`	Nom du jour de la semaine en entier
`%b`	Nom du mois en abrégé
`%B`	Nom du mois en entier
`%c`	Date et heure (par exemple : 10:41:50. 30-Jun-_95)
`%d`	Jour du mois compris entre 1 et 31
`%H`	Heure comprise entre 00 et 23
`%I`	Heure comprise "à l'américaine, AM et PM", entre 00 et 11 (à l'américaine, AM et PM)
`%j`	Le jour de l'année compris entre 001 et 366

Tableau 19.1 : Spécificateurs de format à utiliser pour strftime() *(suite)*

Spécificateur	*Remplacé par*
%m	Le numéro du mois compris entre 01 et 12
%M	La nombre de minutes compris entre 00 et 59
%p	L'une des deux chaînes AM ou PM
%S	Le nombre de secondes compris entre 00 et 59
%U	Le numéro de la semaine de l'année compris entre 00 et 53. Dimanche est considéré comme le premier jour de la semaine
%w	Le jour de la semaine compris entre 0 et 6 (dimanche = 0)
%W	Comme %U, mais, ici, c'est lundi qui est considéré comme le premier jour de la semaine
%x	La date représentée sous forme "locale", par e. Exemples : 09/22/95 (Visual C++, version 2.0) ou Fri Sep 22, 1995 (Borland C++, version 4.02)
%X	L'heure, sous la forme HH:MM:SS
%y	Les deux derniers chiffres de l'année (par exemple : 95)
%Y	L'année "complète" (par exemple : 1995)
%Z	Le nom de la zone horaire ou son abréviation, ou rien si elle ne peut pas être déterminée (par exemple : EDT pour Borland ; rien pour Microsoft)
%%	Le caractère % lui-même

Calcul d'une différence de temps

La fonction `difftime()` permet de calculer la différence entre deux valeurs de type `time t` et renvoie leur différence. Son prototype est :

```
double difftime(time_later, time_earlier);
```

(C'est-à-dire, respectivement : "temps le plus récent", "temps le plus ancien".) La fonction renvoie le nombre de secondes séparant les deux valeurs. Nous en verrons un exemple dans le programme du Listing 19.2.

Vous pouvez déterminer la durée en *tops d'horloge*, c'est-à-dire en intervalles de résolution de l'horloge interne de votre ordinateur en appelant la fonction `times()` dont le prototype est :

```
clock_t times(struct tms *buf);
```

Il faut prendre une mesure au début du programme et faire la différence avec la mesure en fin de programme. Le résultat est exprimé non pas en ticks mais en tops d'horloge. Le nombre de tops d'horloge par seconde s'obtient avec `sysconf( SC CLK TCK)`.

Utilisation des fonctions relatives au temps

Le programme du Listing 19.2 illustre la façon d'utiliser les fonctions relatives au temps de la bibliothèque standard.

Listing 19.2 : Utilisation des fonctions de temps de la bibliothèque standard

```
/* Exemples d'utilisation des fonctions de temps. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <unistd.h>

int main()
{
   time_t start, finish, now;
   struct tm *ptr;
   char *c, buf1[80];
   double duration;
   clock_t top_start, top_finish;
   struct tms buf;

   /* Heure de début de l'exécution. */

   start = time(0);
   top_start = times(&buf);

   /* Appel de ctime() pour enregistrer l'instant de
      début du programme. */

   time(&now);

   /* Convertir la valeur time en une structure de type tm. */

   ptr = localtime(&now);

   /* Créer et afficher une chaîne de caractères contenant
      l'heure actuelle. */

   c = asctime(ptr);
   puts(c);
   getc(stdin);

   /* Utiliser maintenant la fonction strftime() pour créer
      plusieurs versions formatées du temps (date/heure) */
   strftime(buf1, 80, "Nous sommes dans la semaine \
```

Listing 19.2 : Utilisation des fonctions de temps de la bibliothèque standard *(suite)*

```
41:                  %U de l'année %Y", ptr);
42:         puts(buf1);
43:         getc(stdin);
44:
45:         strftime(buf1, 80, "Aujourd'hui, nous sommes %A, %x", ptr);
46:         puts(buf1);
47:         getc(stdin);
48:
49:         strftime(buf1, 80, "Il est %H heures et %M minutes.", ptr);
50:         puts(buf1);
51:         getc(stdin);
52:
53:         /* Prenons l'heure courante pour obtenir la durée
54:            d'exécution du programme. */
55:         finish = time(0);
56:         top_finish = times(&buf);
57:         duration = difftime(finish, start);
58:         printf("\nDurée d'exécution du programme en utilisant \
59:     time() = %f secondes.",duration);
60:         /* Affichons la même durée, mais calculée avec times(). */
61:
62:         printf("\nDurée d'exécution du programme en utilisant \
63:         clock() = %ld centièmes de seconde.",
64:         100 * (top_finish - top_start)/sysconf(_SC_CLK_TCK));
65:         exit(EXIT_SUCCESS);
66:   }
```

Résultats obtenus avec le compilateur GCC 4.1.3 sur Linux (noyau 6.2) :

```
Thu Jan 10 20:32:11 2008

Nous sommes dans la semaine     01 de l'année 2008
Aujourd'hui, nous sommes Thursday, 01/10/08
Il est 20 heures et 32 minutes.

Durée d'exécution du programme en utilisant time() = 5.000000 secondes;
Durée d'exécution du programme en utilisant clock() = 5980 centièmes de seconde.
```

Analyse

Il y a beaucoup de lignes de commentaires dans ce programme, ce qui dispense les auteurs d'en rajouter ici. L'instant de début du programme est enregistré par l'appel à `time()` de la ligne 19. Ce qui va être compté, ce n'est pas le temps d'exécution du programme proprement dit, mais les temps d'attente occasionnés par les `getc()` présents aux lignes 36, 43, 47 et 51. Autrement dit, le temps de lecture de l'écran par l'utilisateur.

On sera sans doute surpris de voir que le numéro de semaine est 1 au lieu de 2. En fait, `%U` renvoie le numéro de semaine dans un intervalle de 0 à 52. Il faut donc ajouter 1 pour avoir un nombre comparable à celui du calendrier des Postes. Outre `%U`, il est aussi possible

d'afficher le numéro de semaine avec `%V` au format ISO 8601:1988 ou avec `%W` où la semaine 0 est celle du premier lundi de l'année.

Fonctions de traitement d'erreur

La bibliothèque standard C contient un certain nombre de fonctions et de macros destinées à vous aider dans le traitement d'erreurs survenant lors de l'exécution d'un programme.

La fonction *assert()*

C'est en réalité une macro, dont le prototype se trouve dans le fichier assert.h , et qui sert principalement à mettre en évidence des bogues dans un programme :

```
void assert(int expression);
```

L'argument *expression* peut être n'importe quelle expression ou variable que vous souhaitez tester. Si le résultat vaut vrai, `assert()` ne fait rien et on passe à l'instruction suivante. Si le résultat vaut faux, `assert()` affiche un message d'erreur sur `stderr` et le programme se termine.

À quelles fins utilise-t-on `assert()` ? Essentiellement pour déceler des erreurs dans un programme au moment de son exécution et non de sa compilation. Par exemple, supposez que vous ayez écrit un programme d'analyse financière qui donne, de temps à autre, des réponses incorrectes. Vous pensez que le problème peut provenir de la variable `taux d interet` prenant une valeur négative (ce qui, financièrement, est désastreux). Pour le vérifier, il suffit de placer "au bon endroit" l'instruction :

```
assert(taux_d_interet >= 0);
```

Si jamais cette variable devient négative, le programme se terminera après avoir affiché un message d'erreur situant l'endroit du programme où était `assert()` et la condition testée.

Le programme du Listing 19.3 permet de voir comment fonctionne cette macro. Si vous lui donnez une valeur non nulle, le programme affichera cette valeur et se terminera normalement. Sinon, il se terminera en erreur avec un message du type :

```
list19_3: list19_3.c:13: main: Assertion `x>0' failed.
```

Attention, vous ne pouvez utiliser cette macro que si votre programme a été compilé en mode "débogué". Vous trouverez comment activer ce mode en consultant la documentation de votre compilateur. Lorsque vous compilerez ensuite la version finale du programme corrigé en mode normal, les macros `assert()` seront désactivées.

Listing 19.3 : Utilisation de la macro assert()

```
/* La macro assert(). */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
    int x;

   printf("Tapez un nombre entier : ");
   scanf("%d", &x);

   assert(x>0);

   printf("Vous avez tapé : %d.\n", x);
   exit(EXIT_SUCCESS);
}
```

Avec les compilateurs Turbo C de Borland et Visual C++ de Microsoft, on obtient les résultats suivants :

```
C:\BC\DIVERS\List19_3
Tapez un nombre entier : 8
Vous avez tapé : 8.

C:\BC\DIVERS\list19_3
Tapez un nombre entier : 0

Assertion failed: x, file C:\BC\DIVERS\LIST19_3.C, line 13
abnormal program termination
```

Sur Linux avec le compilateur GCC, le résultat est similaire :

```
$ list19_3
Tapez un nombre entier : 8
Vous avez tapé : 8.

$ list19_3
Tapez un nombre entier : 0
avirer: avirer3.c:13: main: Assertion `x>0' failed.
Abandon
```

Ce message pourra être différent selon le système et le compilateur utilisés.

Avec le compilateur Borland C++ 4.02, tournant sous Windows 95, l'exécution s'effectue dans une fenêtre MS-DOS. Mais une fois qu'on a tapé 0, le premier message d'erreur apparaît dans une fenêtre d'erreur superposée à la fenêtre MS-DOS. Puis, dès qu'on a cliqué sur le bouton OK, la fenêtre est remplacée par une autre, dans laquelle on lit : "Program Aborted".

Analyse

L'action de `assert()` dépend d'une constante symbolique appelée `NDEBUG`. Si elle n'est pas définie (option par défaut), `assert()` est active. Si on a écrit :

```
#define NDEBUG
#include <assert.h>
.....
```

elle devient inactive. Il est donc simple de placer un peu partout des appels à `assert()` et, une fois la mise au point d'arrêt achevée, sans y toucher, de les désactiver par le `#define NDEBUG` que nous venons de voir.

Il est inutile de donner une valeur particulière à la suite du `#define`. Nous verrons pourquoi au Chapitre 21.

Le fichier d'en-tête errno.h

Le fichier d'en-tête errno.h définit plusieurs macros servant à définir et documenter les erreurs intervenant à l'exécution d'un programme. Ces macros sont utilisées en conjonction avec la fonction `perror()` que nous allons décrire un peu plus loin.

Dans errno.h, on trouve aussi la déclaration d'une variable externe appelée `errno`. Certaines fonctions de la bibliothèque C peuvent lui assigner une valeur lorsqu'une erreur survient en cours d'exécution. Nous avons déjà rencontré cette variable au Chapitre 17 lorsqu'il s'agissait de vérifier la validité d'un résultat de la fonction `strtol()`. errno.h contient aussi un groupe de constantes symboliques correspondant à ces erreurs. Le Tableau 19.2 en présente quelques-unes.

Tableau 19.2 : Quelques-unes des constantes symboliques définies dans errno.h

Nom	*Valeur*	*Message et signification*
`E2BIG`	20	Liste d'arguments trop longue (> 128 octets)
`EACCESS`	5	Permission refusée (par exemple, après une tentative d'écriture sur un fichier ouvert en lecture seule)
`EBADF`	6	Mauvais descripteur de fichier
`EDOM`	33	Argument mathématique en dehors du domaine autorisé
`EEXIST`	35	Le fichier existe déjà
`EMFILE`	4	Trop de fichiers ouverts
`ENOENT`	2	Fichier ou répertoire inexistant

Tableau 19.2 : Quelques-unes des constantes symboliques définies dans errno.h *(suite)*

Nom	*Valeur*	*Message et signification*
ENOEXEC	21	Erreur sur le format d'exécution
ENOMEM	8	Mémoire suffisante
ENOPATH	3	Chemin d'accès non trouvé
ERANGE	34	Résultat en dehors des limites

Il y a deux façons d'utiliser errno. Certaines fonctions signalent, au moyen de leur valeur de retour, qu'une erreur vient de se produire. Dans ce cas, vous pouvez tester la valeur de errno pour déterminer la nature de l'erreur et exécuter telle ou telle action. Sinon, lorsque rien ne vient spontanément vous signaler qu'une erreur a eu lieu, testez errno. Si sa valeur n'est pas nulle, c'est qu'une erreur est survenue et cette valeur indique la nature de l'erreur. N'oubliez pas de remettre errno à zéro après avoir traité l'erreur. Lorsque nous aurons étudié perror(), vous pourrez voir sur le Listing 19.4 un exemple d'utilisation d'errno.

La fonction *perror()*

La fonction perror() est un autre spécimen des outils que C propose pour le traitement des erreurs. Lorsqu'elle est appelée, cette fonction envoie sur stderr un message décrivant la plus récente erreur survenue pendant l'appel d'une fonction de la bibliothèque ou d'une fonction système. Si vous appelez perror() en l'absence de toute erreur, le message sera no error.

Un appel à perror() n'effectue aucun traitement de l'erreur, la fonction se contentant d'envoyer un message. C'est au programme de décider de l'action à entreprendre. Celle-ci peut consister à demander à l'utilisateur d'arrêter le programme. Il n'est pas nécessaire que le programme contienne un #include <errno.h> pour pouvoir utiliser errno. Ce fichier d'en-tête n'est indispensable que si vous voulez utiliser les constantes symboliques figurant dans le Tableau 19.2. Le Listing 19.4 montre l'utilisation de la fonction perror() et de errno pour le traitement des erreurs d'exécution.

Listing 19.4 : Utilisation de perror() et de errno pour traiter les erreurs survenant à l'exécution

```
/* Exemple de traitement d'erreur avec perror()et errno. */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

```

```
int main()
{
     FILE *fp;
    char filename[80];

    printf("Indiquez un nom de fichier : ");
    lire_clavier(filename, sizeof(filename));

    if ((fp = fopen(filename, "r")) == NULL)
    {
        perror("Vous vous êtes trompé !");
        printf("errno = %d.\n", errno);
        exit(EXIT_FAILURE);
    }
    else
    {
        puts("Fichier ouvert en lecture.");
        fclose(fp);
    }
    exit(EXIT_SUCCESS);
}
```



```
Indiquez un nom de fichier : toto
Fichier ouvert en lecture.

Indiquez un nom de fichier : zozor
Vous vous êtes trompé !: No such file or directory
errno = 2.
```

Dans les pays anglo-saxons, l'effet est, sans doute, assez réussi. Dans notre pays, ce mélange de langues fait un peu désordre.

Analyse

Le programme peut afficher deux messages, selon le fichier qu'on veut ouvrir. Si on peut l'ouvrir en lecture, tout va bien ; sinon (fichier inexistant), on affiche un message d'erreur composé d'une partie sur mesure ("Vous vous êtes trompé") à laquelle `perror()` concatène son propre message (": no such file or directory").



À faire

Inclure le fichier d'en-tête errno.h si vous avez l'intention d'utiliser les constantes symboliques d'erreur dont quelques-unes sont listées au Tableau 19.2.

Rechercher d'éventuelles erreurs dans le programme. Ne supposez jamais que tout va pour le mieux dans le meilleur des mondes.

À ne pas faire

Inclure le fichier d'en-tête errno.h si vous n'avez pas l'intention d'utiliser les constantes symboliques d'erreur.

Utiliser la valeur des constantes du Tableau 19.2. Si vous avez besoin de tester errno, comparez-le aux constantes symboliques.

Recherche et tri

Parmi les tâches les plus courantes qu'un programme peut avoir à accomplir, on trouve la recherche (consultation de table ou de liste) et le tri. La bibliothèque standard du C contient des fonctions d'usage général pour ces deux tâches.

Recherche avec *bsearch()*

La fonction de bibliothèque `bsearch()` (de l'anglais *Binary SEARCH*) accomplit une recherche dichotomique dans un tableau d'objets afin d'y trouver une correspondance avec une clé de recherche donnée. Le tableau doit être préalablement trié en ordre croissant. Le programme doit fournir une fonction de comparaison capable de renvoyer un entier négatif, positif ou nul, selon que le résultat de la comparaison avec la clé de recherche est inférieur, égal ou supérieur. Son prototype se trouve dans stdlib.h :

```
void *bsearch(void *key, void *base, size_t num, size_t width,
              int (*cmp)(void *element1, void *element2));
```

Ce prototype plutôt complexe mérite d'être étudié minutieusement :

- `key` est un pointeur vers l'argument de recherche (la clé).
- `base` est un pointeur vers le premier élément du tableau de recherche.
- `num` est le nombre d'éléments du tableau.
- `width` est la taille d'un élément.
- `cmp` est un pointeur vers la fonction de comparaison.

Les deux premiers pointeurs sont de type `void`. Cela permet de faire une recherche dans un tableau contenant n'importe quel type d'objets.

`num` et `width` sont de type `size_t`, car leur valeur est généralement obtenue par un appel à l'opérateur `sizeof()`.

La fonction `cmp` est généralement une fonction écrite par l'utilisateur. Si la recherche s'effectue sur des chaînes de caractères, on utilise la fonction de bibliothèque standard `strcmp()`. Elle doit répondre au cahier des charges suivant :

- Elle reçoit en argument deux pointeurs, un sur chacun des objets à comparer.
- Elle renvoie un entier :
  - négatif si `element1 < element2` ;.
  - nul si `element1 = element2` ;.
  - positif si `element1 > element2`..

La valeur de retour de `bsearch()` est un pointeur de type `void` vers le premier élément du tableau qui soit égal à la clé de recherche. S'il n'y a pas de correspondance, on obtient `NULL`.

L'algorithme de recherche dichotomique est très efficace. Nous avons vu qu'il exigeait que le tableau soit préalablement trié en ordre ascendant. Voici comment il opère :

1. La clé est comparée à l'élément situé au milieu du tableau. Selon le signe du résultat de la comparaison, on sait que la clé se trouve en dessous ou au-dessus du point de correspondance. Si la fonction de comparaison renvoie zéro, cet élément constitue la réponse cherchée.
2. La recherche se poursuit dans l'une des deux moitiés qui est, à son tour, divisée en deux.
3. Et ainsi de suite jusqu'à ce qu'on ait trouvé une correspondance, ou qu'il ne reste plus d'éléments.

Cette méthode élimine à chaque coup une moitié du tableau restant. Dans un tableau de mille éléments, on trouvera le résultat en dix comparaisons, au plus. En général, pour un tableau de 2^n éléments, il faut `n` comparaisons.

Tri avec qsort()

La fonction de bibliothèque standard `qsort()` est une implémentation de l'algorithme de tri *quicksort*, inventé par C. A. R. Hoare. Le tableau est généralement trié en ordre croissant, mais il est possible de le trier en ordre décroissant. Le prototype de la fonction (qui est définie dans stdlib.h) est le suivant :

```
void qsort(void *base, size_t num, size_t size,
           int (*cmp)(void *element1, void *element2));
```

L'argument `base` pointe sur le premier élément du tableau de `num` éléments d'une taille de `size` octets chacun ; `cmp` est un pointeur vers une fonction de comparaison analogue à celle utilisée pour `bsearch()`. La fonction `qsort()` ne renvoie aucune valeur.

Recherche et tri : deux exemples

Le programme du Listing 19.5 montre l'utilisation de `qsort()` et `bsearch()`. Vous noterez la présence, dans le programme, de la fonction `getc()` destinée à permettre de voir ce qui se passe en arrêtant temporairement l'exécution du programme.

Listing 19.5 : Utilisation de qsort() et bsearch() pour trier des valeurs

```
/* Utilisation de qsort()et de bsearch() avec des valeurs.*/

#include <stdio.h>
#include <stdlib.h>

#define MAX 20

int intcmp(const void *v1, const void *v2);

int main()
{
    int arr[MAX], count, key, *ptr;

    /* L'utilisateur va taper quelques nombres entiers. */

    printf("Tapez %d valeurs entières séparées par un \
            appui sur Entrée.\n", MAX);
    for (count = 0; count < MAX; count++)
        scanf("%d", &arr[count]);

    puts("Appuyez sur Entrée pour effectuer le tri.");
    getc(stdin);

    /* Trier le tableau en ordre croissant. */

    qsort(arr, MAX, sizeof(arr[0]), intcmp);

    /* Afficher le tableau trié. */

    for (count = 0; count < MAX; count++)
        printf("\narr[%d] = %d.", count, arr[count]);

    puts("\nAppuyez sur Entrée pour continuer.");
    getc(stdin);

    /* Entrée d'une clé de recherche. */

    printf("Tapez la valeur à rechercher : ");
    scanf("%d", &key);

```

```
    /* Effectuer la recherche. */

    ptr = (int *)bsearch(&key, arr, MAX, sizeof(arr[0]),intcmp);

    if (ptr != NULL)
        printf("%d trouvé à arr[%d].", key, (ptr - arr));
    else
        printf("%d non trouvé.", key);
    exit(EXIT_SUCCESS);
}

int intcmp(const void *v1, const void *v2)
{
    return (*(int *)v1 - *(int *)v2);
}
```

```
Tapez 20 valeurs entières séparées par un appui sur Entrée.
45
12
999
1000
321
123
2300
954
1968
12
2
1999
3261
1812
743
1
10000
3
76
329
Appuyez sur Entrée pour effectuer le tri.

arr[0] = 1.
arr[1] = 2.
arr[2] = 3.
arr[3] = 12.
arr[4] = 12.
arr[5] = 45.
arr[6] = 76.
arr[7] = 123.
arr[8] = 321.
arr[9] = 329.
arr[10] = 743.
arr[11] = 954.
arr[12] = 999.
arr[13] = 1000.
arr[14] = 1812.
arr[15] = 1968.
```

```
arr[16] = 1999.
arr[17] = 2300.
arr[18] = 3261.
arr[19] = 10000.
Appuyez sur Entrée pour continuer.

Tapez la valeur à rechercher : 1812
1812 trouvé à arr[14].
```

Analyse

Le programme commence par vous demander de taper un nombre `MAX` de valeurs (ici, 20). Il les trie en ordre croissant et les affiche dans cet ordre. Ensuite, il vous demande une valeur de recherche et affiche le rang de l'élément où il l'a trouvée ou "non trouvée". La logique du programme est absolument séquentielle, ce qui fait qu'avec les explications données plus haut, tout commentaire serait redondant.

Le Listing 19.6 illustre l'emploi de ces fonctions, cette fois sur des chaînes de caractères.

Listing19.6 : Utilisation de qsort() et bsearch() pour trier des chaînes de caractères

```
/* Utilisation de qsort()et de bsearch() sur des chaînes de
   caractères. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 20

int comp(const void *s1, const void *s2);

int main()
{
    char *data[MAX], buf[80], *ptr, *key, **key1;
    int count;

    /* Entrée d'une suite de mots. */

    printf("Tapez %d mots séparés par un appui sur \
Entrée.\n", MAX);
    for (count = 0; count < MAX; count++)
    {
        printf("Mot %d : ", count+1);
        lire_clavier(buf, sizeof(buf));
        data[count] = strdup(buf);
    }

    /* Trier les mots (en réalité, les pointeurs). */

    qsort(data, MAX, sizeof(data[0]), comp);
```

```

    /* Afficher les mots triés. */

    for (count = 0; count < MAX; count++)
        printf("\n%d: %s", count+1, data[count]);

    /* Demander une clé de recherche. */

    printf("\n\nTapez une clé de recherche : ");
    lire_clavier(buf, sizeof(buf));

    /* Effectuer la recherche. Commencer par définir key1 comme
       un pointeur vers le pointeur sur la clé de recherche */

    key = buf;
    key1 = &key;
    ptr = bsearch(key1, data, MAX, sizeof(data[0]), comp);

    if (ptr != NULL)
        printf("%s trouvé.\n", buf);
    else
        printf("%s non trouvé.\n", buf);
    exit(EXIT_SUCCESS);
}

int comp(const void *s1, const void *s2)
{
    return (strcmp(*(char **)s1, *(char **)s2));
}
```

```
Tapez 20 mots séparés par un appui sur Entrée.
Mot 1 :  abricot
Mot 2 :  amande
Mot 3 :  ananas
Mot 4 :  banane
Mot 5 :  brugnon
Mot 6 :  chou
Mot 7 :  concombre
Mot 8 :  courgette
Mot 9 :  framboise
Mot 10 : groseille
Mot 11 : laitue
Mot 12 : mûre
Mot 13 : orange
Mot 14 : oseille
Mot 15 : poire
Mot 16 : pomme
Mot 17 : pomme
Mot 18 : potiron
Mot 19 : pêche
Mot 20 : raisin
```

```
 1:  abricot
 2:  amande
 3:  ananas
 4:  banane
 5:  brugnon
 6:  chou
 7:  concombre
 8:  courgette
 9:  framboise
10: groseille
11: laitue
12: mûre
13: orange
14: oseille
15: poire
16: pomme
17: pomme
18: potiron
19: pêche
20: raisin

Tapez une clé de recherche : potiron
potiron trouvé.
```

Analyse

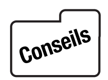
L'organisation générale du programme est identique à celle du précédent, à quelques détails près. On fait ici usage d'un tableau de pointeurs vers les chaînes de caractères, technique qui vous a été présentée au Chapitre 15. Comme nous l'avons vu, on peut trier des chaînes en ne triant que leurs pointeurs. Il faut, pour cela, modifier la fonction de comparaison. On lui passera un pointeur vers chacun des pointeurs des éléments du tableau à comparer. Faute de quoi, ce seraient les pointeurs qui seraient triés et non les éléments sur lesquels ils pointent.

À l'intérieur de la fonction, il est alors nécessaire de "déréférencer" les pointeurs pour atteindre chaque élément. C'est la raison du double astérisque de la ligne 57.

La clé de recherche a été placée en `buf[]` et on sait que le nom d'un tableau (ici, `buf`) est un pointeur vers le tableau. Mais ce qu'on veut passer, ce n'est pas `buf` lui-même, mais un pointeur vers `buf`. Seulement, `buf` est une constante pointeur, et non une variable pointeur, et n'a donc pas d'adresse en mémoire. C'est pourquoi on ne peut pas créer un pointeur qui pointe vers `buf` au moyen de l'opérateur *adresse* devant `buf` en écrivant `&buf`.

Que peut-on faire ? D'abord, créer une variable pointeur et lui assigner la valeur de `buf` (ligne 45). Dans le programme, elle s'appelle `key`. Comme `key` est une variable, elle a une adresse et vous pouvez créer un pointeur contenant cette adresse. Nous l'appellerons `key1` (ligne 46).

Lors de l'appel de `bsearch()`, le premier argument est `key1`, qui est un pointeur vers un pointeur vers la chaîne de caractères qui est la clé. (Ouf !)



À ne pas faire

Oublier de trier votre tableau en ordre croissant avant d'appeler `bsearch()`.

Oublier de libérer la mémoire allouée (ici avec avec `strdup()`).

Résumé

Dans ce chapitre, nous avons exploré quelques-unes des fonctions les plus utiles de la bibliothèque standard du C. Certaines font des calculs mathématiques, d'autres traitent du temps, d'autres encore vous assistent dans la mise au point de vos programmes et le traitement des erreurs. Les fonctions de tri et de recherche sont particulièrement utiles, car elles permettent d'économiser beaucoup de temps lors de l'écriture de vos programmes. Sachez qu'il existe par ailleurs des bibliothèques de fonctions qui implémentent des structures de données plus adaptées à la recherche ou au tri des données. Si vous devez utiliser une table de hachage ou un arbre binaire balancé, ne réinventez pas la roue.

Q & R

Q Pourquoi est-ce que la plupart des fonctions mathématiques renvoient un double ?

R Pour une question de précision, car on obtient ainsi davantage de chiffres significatifs et un exposant plus étendu qu'avec de simples float.

Q Est-ce que `bsearch()` et `qsort()` sont les seules façons de chercher et de trier en C ?

R Bien sûr que non. Elles figurent dans la bibliothèque standard pour votre commodité, mais vous n'êtes pas obligé de les utiliser. Dans beaucoup de livres sur le C vous trouverez des algorithmes de tri et de recherche et les programmes qui vont avec. Ces fonctions sont d'ailleurs déjà implémentées dans des bibliothèques non standard. Certaines sont néanmoins assez répandues (comme glib sur Linux, Windows, MacOS X...) pour que vous puissiez les utiliser tout en garantissant à votre programme une certaine portabilité. Le plus grand intérêt des fonctions `bsearch()` et `qsort()` est qu'elles sont déjà prêtes et qu'elles sont fournies avec tous les compilateurs ANSI.

Q Est-ce que les fonctions mathématiques vérifient les arguments qui leur sont passés ?

R Absolument pas. C'est à vous de le faire. Si, par exemple, vous passez une valeur négative à `sqrt()`, vous obtiendrez une erreur.

Atelier

L'atelier vous propose quelques questions permettant de tester vos connaissances sur les sujets que nous venons d'aborder dans ce chapitre.

Quiz

1. Quel est le type de la valeur renvoyée par quasi toutes les fonctions mathématiques ?
2. À quel type de variable C le type `time t` est-il équivalent ?
3. Quelles sont les différences entre les fonctions `time()`, `times()` et `clock()` ?
4. Lorsque vous appelez la fonction `perror()`, que fait-elle pour corriger une condition d'erreur ?
5. Avant de pratiquer une recherche dans un tableau avec `bsearch()`, que devez-vous faire ?
6. Avec `bsearch()`, combien de comparaisons vous faudra-t-il, au plus, pour trouver une correspondance dans un tableau de 16 000 articles ?
7. Avec `bsearch()`, combien de comparaisons vous faudra-t-il, au plus, pour trouver une correspondance dans un tableau de 10 articles ?
8. Avec `bsearch()`, combien de comparaisons vous faudra-t-il, au plus, pour trouver une correspondance dans un tableau de 2 millions d'articles ?
9. Quelle valeur doit renvoyer la fonction de comparaison figurant en argument de `bsearch()` et `qsort()` ?
10. Que renvoie `bsearch()` si elle ne trouve pas de correspondant dans le tableau ?

Exercices

1. Écrivez un appel à `bsearch()`. Le tableau dans lequel s'effectuera la recherche s'appelle `names` et contient des chaînes de caractères. La fonction de comparaison s'appelle `comp names()`. On supposera que tous les noms ont la même longueur.
2. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions qui suivent ?

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ int value[10], count, key, *ptr;

  printf("Tapez des valeurs :");
  for (ctr=0; ctr<10; ctr++)
    scanf("%d", &values[ctr]);
```

```
    qsort(values, 10, compare_function());
    exit(EXIT_SUCCESS);
  }
```

3. **CHERCHEZ L'ERREUR :** Y a-t-il quelque chose de faux dans les instructions qui suivent ?

```
  int intcmp(int element_1, int element_2)
  { if (element_1 > element_2) return -1;
    if (element_1 < element_2) return 1;
    return 0;
  }
```

   On ne donne pas les corrigés des exercices suivants.

4. Modifiez le programme du Listing 19.1 pour que la fonction `sqrt()` puisse travailler sur la valeur absolue des nombres négatifs.
5. Écrivez un programme qui consiste en un menu vous proposant plusieurs fonctions mathématiques. Mettez-y autant de fonctions que vous pourrez.
6. Écrivez une fonction qui arrête le programme pendant environ 5 secondes, en utilisant les fonctions de traitement du temps que nous avons vues dans ce chapitre.
7. Ajoutez la fonction `assert()` au programme de l'exercice 4 de façon à ce qu'il puisse afficher un message lorsque l'utilisateur tape un nombre négatif.
8. Écrivez un programme qui lit 30 noms tapés par l'utilisateur et les trie avec `qsort()`. Il devra afficher les noms une fois triés.
9. Modifiez le programme de l'exercice 8 pour que, si l'utilisateur tape "quit", le programme cesse de demander des noms et se mette à trier ceux qu'il a déjà reçus.
10. Au Chapitre 15, vous avez vu une méthode de tri élémentaire dont nous vous avons signalé la lenteur. Faites-lui trier un grand tableau puis comparez le temps qu'elle a mis pour le faire au temps mis par la fonction de bibliothèque `qsort()` pour trier le même tableau.

Exemple pratique 6

Calcul des versements d'un prêt

Le programme présenté dans cette section est destiné au calcul des remboursements d'un emprunt. En l'exécutant, vous devrez lui transmettre trois informations :

- Le montant de l'emprunt (ou principal).
- Le taux d'intérêt annuel. Vous devez indiquer le taux réel ; par exemple, 8,5 pour 8,5 %. Ne calculez pas la valeur numérique réelle (0,085 dans notre cas) puisque le programme s'en charge.
- Le nombre de mensualités pour le remboursement.

Ce programme vous permettra de calculer le tableau d'amortissement d'un prêt immobilier ou de tout autre type d'emprunt.

Listing Exemple pratique 6 : Calcul du montant des remboursements d'un prêt

```
/* Calcul des mensualités d'un emprunt. */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main()
{
    float principal, rate, payment;
   int term;
   char ch;

   while (1)
   {
```

```
15:         /* Lecture des données concernant l'emprunt */
16:         puts("\nEntrez le montant de l'emprunt: ");
17:         scanf("%f", &principal);
18:         puts("\nEntrez le taux d'intérêt annuel: ");
19:         scanf("%f", &rate);
20:         /* Ajustement du pourcentage. */
21:         rate /= 100;
22:         /* Calcul du taux d'intérêt mensuel. */
23:         rate /= 12;
24:
25:         puts("\nEntrez la durée de remboursement du prêt en mois: ");
26:         scanf("%d", &term);
27:         payment = (principal * rate) / (1 - pow((1 + rate), -term));
28:         printf("Vos mensualités se monteront à $%.2f.\n", payment);
29:
30:         puts("Autre calcul (o ou n)?");
31:         do
32:         {
33:             ch = getchar();
34:         } while (ch != 'n' && ch != 'o');
35:
36:         if (ch == 'n')
37:             break;
38:     }
39:     exit(EXIT_SUCCESS);
40: }
```

Analyse

Ce programme de calcul est prévu pour un prêt standard comme le financement d’une voiture à taux fixe ou un emprunt immobilier. Les remboursements sont calculés à l’aide de la formule financière suivante :

```
paiement = (P * R) / (1 - (1 + R)^(-T))
```

`P` est le principal, `R` le taux d’intérêt, et `T` la période. Le symbole `^` signifie "à la puissance". Dans cette formule, il est important d’exprimer la période et le taux avec la même unité de temps. Si la durée de l’emprunt est indiquée en mois, par exemple, le taux d’intérêt devra aussi être le taux mensuel. Les taux d’intérêts étant généralement exprimés en taux annuels, la ligne 23 divise ce taux par 12 pour obtenir le taux mensuel correspondant. Le calcul des échéances s’effectue en ligne 27 et la ligne 28 affiche les résultats.

20

La mémoire

Ce chapitre traite quelques aspects parmi les plus avancés de la gestion de mémoire dans les programmes C :

- Conversions de types
- Allocation et libération de mémoire
- Manipulations sur des blocs de mémoire
- Manipulations des bits

Conversions de types

Tous les objets C ont un type défini. Une variable numérique peut être un `int` ou un `float`, un pointeur peut pointer vers un `double` ou un `char`, et ainsi de suite. Dans les programmes, on a souvent besoin de mélanger différents types dans une même expression. Qu'arrive-t-il alors ? Parfois, C se charge automatiquement des conversions nécessaires ; à d'autres moments, c'est à vous d'effectuer ces conversions pour éviter d'obtenir des résultats erronés. Vous en avez vu des exemples dans les chapitres précédents, et nous avons même étudié le *casting* nécessaire d'un pointeur de type `void` vers un type spécifique de données. Dans ce cas et dans les autres, vous devez comprendre ce qui se passe pour être à même de décider s'il faut ou non effectuer une conversion explicite et d'évaluer les risques d'erreur si vous n'en faisiez pas.

Conversions automatiques de types

Comme leur nom l'indique, ce sont des conversions effectuées automatiquement par le compilateur C, sans que vous ayez à intervenir. Pour juger de leur bien fondé, vous devez comprendre comment C évalue les expressions.

Promotion de type dans une expression

Lorsqu'une expression C est évaluée, la valeur qui en résulte est d'un type particulier. Si tous les constituants de l'expression sont du même type, le résultat est lui-même de ce type. Par exemple, si `x` et `y` sont des `int`, le résultat de l'évaluation de l'expression suivante est aussi du type `int` :

```
x + y
```

Que va-t-il se passer si l'une des variables n'est pas du même type ? Dans ce cas, le compilateur va "s'arranger" pour éviter une perte d'informations en allant du plus "pauvre" vers le plus "riche" dans la liste suivante : `char`, `int`, `long`, `float` et `double`. Donc, si, dans l'expression, `y` était un `char`, le résultat serait du type `int`. Si on associe un `long` et un `float` dans une expression, le résultat sera de type `float`.

À l'intérieur d'une expression, les opérandes individuels sont *promus*, si c'est nécessaire, pour correspondre aux opérandes de l'expression auxquels ils sont associés. Les opérandes subissent cette promotion par paires, pour chaque opérateur binaire. Bien entendu, si deux opérandes sont de même type, aucune promotion n'est nécessaire. Voici les règles suivies, dans cet ordre, par ce mécanisme de promotion :

- Si l'un des opérandes est un `double`, l'autre opérande est promu au type `double`.

- Si l'un des opérandes est un `float`, l'autre opérande est promu au type `float`.
- Si l'un des opérandes est un `long`, l'autre opérande est promu au type `long`.

Si, par exemple, `x` est un `int` et `y` un `float`, l'évaluation de l'expression `x / y` entraînera la promotion de `x` en `float` avant que ne soit effectuée la division. Cela ne signifie pas que le type de `x` a été changé mais, tout simplement, qu'une copie de `x` a été convertie en `float` avant d'effectuer la division. Le résultat de celle-ci est naturellement de type `float`. De la même façon, si `x` est un `double` et `y` un `float`, `y` sera promu en `double`.

Conversion par affectation

Les promotions interviennent de chaque côté du signe (=). Une fois évaluée, l'expression de droite est toujours promue au type de la *L Value* de gauche. Notez que cette opération peut aboutir à une "dégradation" du résultat, c'est-à-dire à sa conversion dans un type plus pauvre. Si, par exemple, `f` est de type `float` et `i` de type `int`, le résultat de l'évaluation de l'expression `i` (c'est-à-dire `i` lui-même) sera promu au type `float` dans l'expression :

```
f = i;
```

Au contraire, si on avait écrit :

```
i = f;
```

c'est `f` qui aurait été converti en `int` par l'ablation de sa partie décimale. Rappelez-vous que cela ne change en rien la valeur de `f` puisque cela ne concerne que la copie de `f` utilisée dans l'affectation. Après avoir exécuté les trois instructions suivantes :

```
float f=1.23;
int i;
i = f;3
```

`i` a la valeur 1 et `f` vaut toujours 1.23.

Lorsqu'un `float` est dégradé en `int`, il subit généralement des pertes alors que, dans l'autre sens, ce n'est pas souvent le cas. En effet, un entier peut toujours être décomposé en une somme de puissances entières de 2 (base de représentation interne dans les machines modernes). Ainsi, dans les instructions suivantes :

```
float f;
int i = 3;
f = i;
printf("%f\n", f);
```

on affichera bien 3. (et non 2.999995). En revanche, dès qu'on opère avec des nombres fractionnaires, les erreurs d'arrondis se cumulent généralement, comme on peut en juger dans l'exemple suivant :

```
#include <stdio.h>

int main()
{ float a=.001, b=0;
  int i;

  for (i=0; i<1000; i++)
   b +=a;

  printf("la somme vaut %8.6f\n", b);
}
```

Le résultat obtenu ne vaut pas 1, mais 0,999991.

On aurait eu d'autres surprises si `i` avait été de type `long`, comme le montre l'exemple suivant :

```
float f;
long i =987654321;
f = i;
printf("%f\n", f);
```

où on affichera comme valeur de `f` : 987654336.000000. Là, l'erreur provient du fait que le nombre de chiffres significatifs d'un `float` est plus petit que celui d'un `long`.

Conversions explicites avec coercition

Nous avons déjà rencontré le *casting*. La dernière fois, c'était au Chapitre 18 où nous avons dit qu'on pouvait traduire ce mot par "coercition". Nous emploierons indifféremment les deux termes, le premier étant préféré par les programmeurs, le second par ceux qui s'attachent à la pureté de leur langue.

Quoi qu'il en soit, nous disposons là du moyen d'effectuer une conversion explicite d'un type dans un autre. L'opérateur de coercition s'écrit en plaçant le type du résultat à obtenir entre parenthèses, devant la variable ou l'expression (entre parenthèses, elle aussi, dans ce cas) à forcer.

Coercition dans les expressions arithmétiques

La coercition oblige le compilateur à effectuer une conversion de type, même (et surtout) s'il n'était pas décidé à le faire implicitement. Par exemple, si `i` est de type `int`, écrire

```
(float)i
```

transforme la copie interne de `i` en `float` (toujours sans changer ce qui se trouve dans la variable `i`).

À quel moment doit-on faire usage de la coercition ? Souvent pour éviter toute perte de précision dans une division, comme on peut le voir dans le programme du Listing 20.1.

Listing 20.1 : Exemple simple d'utilisation de la coercition

```
 1:   #include <stdio.h>
 2:   #include <stdlib.h>
 3:   int main()
 4:   {
 5:     int i1 = 100, i2 = 40;
 6:     float f1, f2;
 7:
 8:     f1 = i1 / i2;
 9:     f2 = (float)i1 / i2;
10:     printf("Sans coercition : %f Avec coercition : %f\n", f1, f2);
11:     exit(EXIT_SUCCESS);
12:   }
```



```
Sans coercition : 2.000000 Avec coercition : 2.500000
```

Analyse

On voit que le premier résultat est grossièrement erroné. En effet, le résultat de la division de deux entiers (ligne 8) est un entier ; donc 100/40 donne 2. Si, en revanche, on emploie la coercition sur l'un des deux opérandes de la division, comme à la ligne 9, il y a conversion implicite de l'autre terme et la division s'effectue entre deux nombres de type `float`. Le résultat rangé dans `f2` est donc un `float`.

Notez que, si on avait écrit :

```
f2 = (float)(i1 / i2);
```

on aurait encore obtenu 2.000000, car c'est le quotient qui aurait subi la coercition, la division ayant été effectuée entre deux `int`.

Mais on aurait pu, aussi bien, user de coercition pour chacun des deux opérandes, plutôt que de laisser faire le compilateur pour l'autre terme, en écrivant :

```
f2 = (float)i1 / (float) i2;
```

Ici, on aurait obtenu 2.500000, valeur correcte.

La coercition appliquée aux pointeurs

Au Chapitre 18, nous avons eu l'occasion de rencontrer la coercition appliquée aux pointeurs. Un pointeur de type `void` est un pointeur générique ne pouvant pointer sur aucun type d'objet défini. Il est donc nécessaire de le *caster* pour pouvoir l'utiliser. Notez qu'il n'est pas nécessaire de le *caster* pour lui assigner une valeur (une adresse), pas plus que pour le comparer à `NULL`. Cependant, vous devez le *caster* avant de l'utiliser pour référencer une variable ou de le faire intervenir dans une opération arithmétique d'addition ou de soustraction (du type `p++`, par exemple).



À faire

Utiliser un casting pour promouvoir ou dégrader une variable, lorsque c'est nécessaire.

À ne pas faire

Utiliser une promotion rien que pour éviter un diagnostic du compilateur. Il faut d'abord bien comprendre la signification de ce diagnostic et voir s'il n'a pas une autre cause.

Allocation d'espace mémoire

La bibliothèque C contient des fonctions d'*allocation de mémoire dynamique*, c'est-à-dire d'allocation de mémoire au moment de l'exécution. Cette technique peut avoir de gros avantages par rapport à la réservation automatique et systématique effectuée au moment de la compilation, qu'on appelle *allocation statique*. Cette dernière demande que les dimensions maximales des tableaux ou structures soient connues au moment de l'écriture du programme, alors que l'allocation dynamique permet de se limiter à la mémoire strictement nécessaire au cas particulier qu'on traite.

Les fonctions à utiliser ont leur prototype dans stdlib.h. Toutes les fonctions d'allocation renvoient un pointeur de type `void`. (Sauf `free()`, mais ce n'est pas, *stricto sensu*, une fonction d'allocation puisque, au contraire, c'est elle qui permet de restituer la mémoire acquise dynamiquement.) Contrairement au langage C++ il ne faut pas *caster* ce pointeur lors de l'appel à une fonction d'allocation. C'est une erreur courante en C qui provient des premières version du langage C et des livres qui se basent dessus sans tenir compte de la norme C89 ; cette erreur est entretenue par la nécessité de *caster* en C++.

Avant d'entrer dans les détails, arrêtons-nous sur la signification physique de cette opération concernant la mémoire dont dispose l'ordinateur (mémoire RAM qui est variable selon la configuration installée). Lorsqu'on exécute un programme quel qu'il soit (un traitement de texte ou un programme écrit par soi-même dans un langage quelconque), il est

chargé à partir du disque dans la mémoire de l'ordinateur. Outre les instructions du programme (et celles des fonctions de la bibliothèque rajoutées au moment de l'édition de liens), on a besoin de place pour loger les variables statiques, c'est-à-dire celles qui ont été déclarées dans le programme. Or, une partie de la mémoire est déjà occupée par diverses composantes du système d'exploitation. On ne peut donc généralement pas savoir s'il restera assez de place, surtout quand on utilise de grands tableaux.

La mémoire est une denrée qui, si elle n'est plus chère et rare comme au temps de MS-DOS, reste précieuse. Cela se sent en particulier pour les exécutables mal programmés, où la mémoire n'est pas systématiquement libérée lorsqu'elle n'est plus nécessaire : ces exécutables se mettent alors à consommer de plus en plus de mémoire, ce qui peut ralentir l'ordinateur de manière significative.

Par ailleurs, si les fonctions d'allocation fonctionnent généralement bien, en vous renvoyant un pointeur vers la zone allouée, vous devez systématiquement tester ce pointeur. S'il est `NULL`, l'allocation a échoué. Lorsque cela arrive, vous pouvez généralement mettre fin à votre programme en affichant un message d'erreur (avec `perror()` par exemple) et en quittant avec `exit(EXIT FAILURE)`. Sans mémoire, point de salut !

La fonction *malloc()*

Au cours des chapitres précédents, vous avez appris à utiliser la fonction `malloc()` pour allouer l'espace nécessaire à des chaînes de caractères. Son utilisation n'est pas limitée à ce type d'objet ; elle est capable d'allouer de la mémoire pour tout objet C. Rappelons que son prototype est :

```
void *malloc(size_t num);
```

L'argument `size t` est défini dans stdlib.h comme un `unsigned`. La fonction renvoie un pointeur sur le premier octet du bloc d'une longueur de `num` octets, ou `NULL` s'il ne reste plus assez de mémoire disponible (ou si `num` vaut 0). Le programme du Listing 20.2 vous montre comment utiliser `malloc()` correctement avec un test vérifiant que la mémoire a été allouée, et avec un appel à `free()` pour libérer la mémoire. Nous verrons `free()` plus loin.

Listing 20.2 : Utilisation classique de malloc()

```
/* Utilisation classique de malloc().*/
#include <stdio.h>
#include <stdlib.h>

/* Définition d'une structure ayant
   une taille de 1024 octets (1 Koctet). */

struct kilo
```

Listing 20.2 : Utilisation classique de malloc() (*suite)*

```
 9:  {
10:    char dummy[1024];
11:  };
12:
13:  int main()14:  {
15:    struct kilo un_kilo;
16:
17:    if(NULL == (un_kilo = malloc(sizeof(*un_kilo))))
18:    {
19:      perror("Problème d'allocation mémoire ");
20:      exit(EXIT_FAILURE);
21:    }
22:    /* La mémoire est allouée.
23:       On peut utiliser un_kilo ici.
24:     */
25:
26:    free(un_kilo);
27:
28:    exit(EXIT_SUCCESS);
29:  }
```

Analyse

Ce programme alloue de la mémoire ligne 17. Vous remarquerez plusieurs points sur cette façon d'allouer la mémoire. Tout d'abord, la taille de la zone à allouer est `sizeof(*un kilo)`, soit la taille du type pointé par le pointeur, soit encore la taille d'un `struct kilo` dans notre cas. En indiquant la taille de cette façon, vous pouvez modifier le type de votre variable. La ligne 17 ne changera pas et correspondra toujours à la bonne taille à allouer. Vous noterez également qu'un test est effectué sur cette même ligne. Cela peut sembler nuire à la lisibilité. En réalité, ce n'est qu'une question d'habitude visuelle. Par contre, c'est également une excellente habitude car en encapsulant l'allocation mémoire dans ce test, vous ne pouvez oublier d'effectuer ce test !

Si l'allocation mémoire échoue, il est inutile de continuer le programme. Il prend fin après avoir affiché un message d'erreur lignes 19 et 20. Sinon, on peut continuer l'exécution qui se termine impérativement par un appel à free(). N'oubliez jamais de libérer la mémoire que vous avez allouée, même si c'est à la fin de votre programme. La raison est qu'un développement ultérieur dans votre programme pourrait faire que ce qui était la fin ne le soit plus. Si vous n'avez pas libéré la mémoire, vous n'y penserez pas forcément en étendant votre programme.

La fonction *calloc()*

La fonction `calloc()` alloue aussi de la mémoire mais, au lieu d'allouer un groupe *d'octets* comme le fait `malloc()`, `calloc()` alloue un groupe *d'objets*. La zone de

mémoire allouée est remise à zéro et la fonction retourne un pointeur vers le premier octet de la zone. Si l'allocation ne peut être satisfaite, `calloc()` se contente de renvoyer `NULL`. Voici son prototype :

```
void *calloc(size_t num, size_t size);
```

On alloue (ou, plus exactement, on tente d'allouer) `num` blocs de `size` octets chacun. Le programme du Listing 20.3 donne un exemple d'utilisation de cette fonction.

Listing 20.3 : Utilisation de calloc() pour allouer dynamiquement de la mémoire

```
/* Utilisation de calloc(). */

#include <stdlib.h>
#include <stdio.h>

int main()
{
    unsigned num;
    int *ptr;

   printf("Indiquez le nombre d'int à allouer : ");
   scanf("%d", &num);

   if(NULL == (ptr = calloc(num, sizeof(*ptr)))
   {
      perror("L'allocation de mémoire n'a pas été possible ");
      exit(EXIT_FAILURE);
   }
   puts("L'allocation de mémoire a réussi.\n");

   free(ptr);
   exit(EXIT_SUCCESS);
}
```



```
Indiquez le nombre d'int à allouer : 10000
L'allocation de mémoire a réussi.
```

Analyse

Ce programme n'a effectué aucune vérification de la valeur entrée par l'utilisateur. Donc :

- Si celui-ci tape une valeur négative, comme elle est rangée dans un `unsigned`, elle "ressemblera" à un nombre positif très grand.
- S'il tape une valeur très grande, supérieure à ce que peut contenir un `unsigned` (par exemple, 99999 sur un PC), la valeur sera tronquée et ce qui sera passé à `calloc()` n'aura rien à voir avec la valeur tapée.

Il ne nous paraît pas utile d'en dire davantage.

La fonction *realloc()*

Cette fonction modifie la taille d'un bloc mémoire précédemment alloué par un appel à `malloc()` ou `calloc()`. Voici son prototype :

```
void *realloc(voit *ptr, size_t size);
```

Le pointeur `ptr` pointe sur le bloc de mémoire original. L'argument `size` indique non pas le supplément de mémoire qu'on veut obtenir, mais la taille totale qu'on veut donner au bloc (inférieure ou supérieure à celle du bloc original). Plusieurs cas sont possibles :

- S'il y a assez de place pour satisfaire la requête, un nouveau bloc est alloué et `ptr` contient son adresse. Le contenu de la zone de mémoire supplémentaire est indéterminé, l'ancien contenu étant inaltéré.
- S'il n'y a pas assez de place, la fonction renvoie `NULL` et le contenu de l'ancien bloc n'est pas altéré.
- Si `ptr` contient `NULL`, `realloc()` se comporte comme `malloc()`.
- Si `size` vaut zéro et que `ptr` ne vaut pas `NULL`, la zone précédemment allouée et pointée par `ptr` est libérée et la fonction renvoie `NULL`.

Le programme du Listing 20.4 montre un exemple simple d'utilisation de `realloc()`.

Listing 20.4 : Utilisation de realloc() pour modifier la taille d'un bloc alloué dynamiquement

```
/* Utilisation de realloc() pour modifier la taille
  d'un bloc alloué dynamiquement. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
     char buf[80], *message;

    /* Entrée d'une chaîne de caractères. */

    puts("Tapez une ligne de texte.");
    lire_clavier(buf, sizeof(buf));

    /* Allouer le bloc initial et y copier la chaîne. */

    message = realloc(NULL, strlen(buf)+1);
    strcpy(message, buf);

    /*Afficher ce qu'on vient de lire au clavier. */

```

```
23:        puts(message);
24:
25:        /* Demander une autre chaîne à l'utilisateur. */
26:
27:        puts("Tapez une autre ligne de texte.");
28:        lire_clavier(buf, sizeof(buf));
29:
30:        /* Augmenter la taille du bloc précédent et concaténer
31:           les deux chaînes de caractères dans ce bloc. */
32:        if(NULL == (
33:           message = realloc(message, (strlen(message) +  strlen(buf)+1)))
34:        {
35:           perror("Erreur de réallocation mémoire ");
36:           exit(EXIT_FAILURE);
37:        }
38:        strcat(message, buf);
39:
40:        /* Afficher la chaîne résultante. */
41:
42:        puts(message);
43:
44:        /* Terminer proprement en libérant la totalité du bloc. */
45:
46:        realloc(message, NULL);
47:        exit(EXIT_SUCCESS);
48:    }
```

Voici un exemple d'exécution :

```
Tapez une ligne de texte.
L'oeil était dans la tombe
L'oeil était dans la tombe
Tapez une autre ligne de texte.
 et regardait Caïn.
L'oeil était dans la tombe et regardait Caïn.
```

Analyse

Le programme demande à l'utilisateur de taper une chaîne de caractères (ligne 13). Une fois que celui s'est exécuté (ligne 14), la chaîne est lue dans un tableau de caractères, `buf`, ayant une longueur fixe de 80 caractères. Cette chaîne est ensuite copiée dans une zone de mémoire acquise dynamiquement (ligne 18). On notera l'utilisation de `realloc()` avec `NULL` en premier argument, équivalente de `malloc()`. La taille de cette zone est juste suffisante pour loger la chaîne de caractères lue.

Après avoir demandé une seconde chaîne de caractères, on appelle `realloc()` en lui passant en arguments le pointeur sur la zone précédemment allouée et la somme des longueurs des deux chaînes (ligne 32). Il ne reste plus qu'à joindre les deux chaînes au moyen de la fonction `strcat()` de la ligne 38, et à libérer la zone (avec une longueur nulle) par le `realloc()` de la ligne 46. Cet exemple illustrait les différents cas que peut

rencontrer la fonction `realloc()`. Bien entendu, préférez `malloc()` (ou `calloc()`) pour allouer un nouvel espace, et `free()` pour libérer la mémoire.

La fonction *free()*

Nous avons déjà rencontré cette fonction de libération de mémoire allouée dynamiquement dans l'exemple du Listing 20.2. Le pool de mémoire dans lequel on effectue des prélèvements pour satisfaire les appels à `malloc()`, `calloc()` ou `realloc()` n'est pas infini ; il convient de restituer ce qui a été acquis dans un programme avant de passer la main au système d'exploitation.

La libération d'un bloc de mémoire s'effectue en appelant la fonction `free()` dont le prototype est le suivant :

```
void free(void *ptr);
```

Attention, cette fonction ne renvoie rien. La mémoire pointée par `ptr` est restituée au pool de mémoire. Si `ptr` vaut `NULL`, la fonction ne fait rien du tout. Le programme du Listing 20.5 illustre son utilisation.

Listing 20.5 : Utilisation de la fonction free() pour restituer de la mémoire acquise dynamiquement

```
/* Utilisation de free() pour libérer
   de la mémoire acquise dynamiquement. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BLOCKSIZE 30000

int main()
{
    void *ptr1, *ptr2;

    /* Allouer un bloc. */

    if(NULL == (ptr1 = malloc(BLOCKSIZE)))
    {
        printf("Il a été impossible d'allouer %d octets.\n",
                BLOCKSIZE);
        exit(EXIT_FAILURE);
    }
    printf("\nPremière allocation de %d octets réussie.",
            BLOCKSIZE);

    /* Essayer d'allouer un autre bloc. */

    if(NULL == (ptr2 = malloc(BLOCKSIZE)))
```

```
27:        {
28:            printf("Le second essai pour allouer %d octets a échoué.\n",
29:                    BLOCKSIZE);
30:            exit(EXIT_FAILURE);
31:        }
32:
33:        /* Si l'allocation réussit, afficher un message ,
34:           libérer les deux blocs et quitter le programme. */
35:
36:        printf("\nSeconde allocation de %d octets réussie.\n",
37:                BLOCKSIZE);
38:
39:        /* Libérer les deux blocs. */
40:        free(ptr1);
41:        free(ptr2);
42:        exit(EXIT_SUCCESS);
58:    }
```

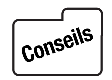


```
Première allocation de 30000 octets réussie.
Seconde allocation de 30000 octets réussie.
```

Analyse

Ce programme va tenter d'allouer dynamiquement deux blocs de `BLOCKSIZE` octets chacun (ici, 30 000). La première allocation s'effectue à la ligne 15 en appelant `malloc()`. Aux lignes 15 à 19, on teste la réussite de l'opération. Si elle échoue, on affiche un message et on s'en va.

On va ensuite tenter d'allouer un second bloc distinct du premier (ligne 26). Si l'opération échoue, inutile de continuer : on affiche un message et on s'en va. Si ces deux opérations ont réussi, on affiche un message, on libère les deux blocs et on s'en va.



À faire

Acquérir de la mémoire et ne pas la libérer quand on n'en a plus besoin est condamnable.

Indiquer la taille de l'espace mémoire à allouer en utilisant `sizeof(*pointeur)`. Le préprocesseur est capable de retrouver le type de l'élément pointé par pointeur et d'en calculer la taille avec `sizeof()`.

À ne pas faire

Supposer qu'un appel à `malloc()`, à `calloc()` ou à `realloc()` est toujours couronné de succès. Vérifiez toujours que la fonction n'a pas renvoyé `NULL`.

Manipulation de blocs de mémoire

Dans la bibliothèque standard C, il existe des fonctions pour manipuler des blocs de mémoire qui permettent de faire des initialisations ou des copies de bloc à bloc, bien plus rapidement, par exemple, qu'avec une boucle `for`.

La fonction *memset ()*

Cette fonction sert à donner à tous les octets d'un bloc de mémoire la même valeur. Son prototype se trouve dans string.h. Il est le suivant :

```
void *memset(void *dest, int c, size_t count);
```

L'argument `dest` pointe sur le bloc de mémoire, `c` est le caractère de garnissage et `count` est le nombre d'octets (de caractères) de la zone qu'on veut initialiser. La valeur de retour est `dest`, ou `NULL` en cas d'erreur.

C'est surtout pour initialiser des blocs de caractères que `memset()` est intéressante. Pour d'autres types de variable, cette fonction ne peut guère être utilisée qu'avec la valeur 0. Nous verrons un exemple d'utilisation de cette fonction dans le Listing 20.6.

La fonction *memcpy()*

Cette fonction copie des blocs d'informations d'un bloc de mémoire dans un autre, sans tenir compte du type. C'est simplement une copie à l'identique, octet par octet. Son prototype se trouve dans string.h. C'est le suivant :

```
void *memcpy(void *dest, const void *src, size_t n);
```

Les arguments `dest` et `src` pointent respectivement vers la zone destinataire et la zone source, et `n` indique le nombre d'octets à copier. La valeur de retour est `dest`. Si les deux blocs se recouvrent, la recopie n'est en général pas correcte, certaines parties de la source pouvant être recouvertes avant d'être copiées. Dans ce cas, il faut utiliser `memmove()`, de même prototype que `memcpy()`.

La fonction *memmove()*

`memmove()` est pratiquement identique à `memcpy()` dont elle constitue un perfectionnement lorsqu'il y a recouvrement des blocs à copier. Son prototype se trouve dans string.h. C'est le suivant :

```
void *memmove(void *dest, const void *src, size_t n);
```

`dest` et `src` pointent respectivement vers la zone destinataire et la zone source, et `n` indique le nombre d'octets à copier. La valeur de retour est `dest`. Même si les deux blocs se recouvrent, la copie s'effectue correctement. Cette fonction devant tenir compte du cas particulier où les zones mémoire se chevauchent, elle est un peu moins rapide que `memcpy()`.

Le Listing 20.6 montre une application des trois fonctions `memset()`, `memcpy()` et `memmove()`.

Listing 20.6 : Exemple d'utilisation de memset(), memcpy() et memmove()

```
/* Exemple d'emploi de memset(), memcpy(), st memmove(). */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char message1[60] = "Le chêne, un jour, dit au roseau";
char message2[60] = "abcdefghijklmnopqrstuvwxyz";
char temp[60];
int main()
{
  printf("\nmessage[l] avant memset():\t%s", message1);
  memset(message1 + 5, '@', 10);
  printf("\nmessage[1] après memset():\t%s", message1);

  strcpy(temp, message2);
  printf("\n\nmessage original : %s", temp);
  memcpy(temp + 4, temp + 16, 10);
  printf("\nAprès memcpy, sans recouvrement :\t%s", temp);
  strcpy(temp, message2);
  memcpy(temp + 6, temp + 4, 10);
  printf("\nAprès memcpy() avec recouvrement :\t%s", temp);

  strcpy(temp, message2);
  printf("\n\nMessage original : %s", temp);
  memmove(temp + 4, temp + 16, 10);
  printf("\nAprès memmove() sans recouvrement :\t%s", temp);
  strcpy(temp, message2);
  memmove(temp + 6, temp + 4, 10);
  printf("\nAprès memmove() avec recouvrement :\t%s\n", temp);
  exit(EXIT_SUCCESS);
}
```

```
message[l] avant memset():Le chêne, un jour, dit au roseau
message[1] après memset():    Le ch@@@@@@@@@@ur, dit au roseau

message original : abcdefghijklmnopqrstuvwxyz
Après memcpy() sans recouvrement :  abcdqrstuvwxyzopqrstuvwxyz
Après memcpy() avec recouvrement :  abcdefefefefefefqrstuvwxyz

Message original : abcdefghijklmnopqrstuvwxyz
Après memmove() sans recouvrement : abcdqrstuvwxyzopqrstuvwxyz
Après memmove() avec recouvrement : abcdefefghijklmnqrstuvwxyz
```

Analyse

Pas de commentaire pour `memset()`. La notation `message1+5` permet de spécifier le point de départ de l'action de `memset()` (6e caractère de `message1`). Il en résulte que les caractères 6 à 15 sont remplacés par des zéros.

Quand il n'y a pas recouvrement (ligne 17), on constate que `memcpy()` fonctionne correctement. En revanche, à l'instruction de la ligne 20, la source étant placée deux positions plus à gauche que la destination, le résultat montre le redoublement des caractères "fe" situés entre les deux points de départ.

Les deux exemples de `memmove()` (lignes 25 et 28) montrent que tout se passe bien dans les deux cas.

Opérations sur les bits

Vous savez que l'unité de base pour le stockage des informations est le bit. Il est quelquefois très pratique de pouvoir manipuler ces bits à partir d'un programme C. À cette fin, ce langage met plusieurs outils à votre disposition.

Vous pouvez manipuler les bits d'une variable entière à l'aide des opérateurs bit à bit. Le bit étant la plus petite unité d'enregistrement, il ne peut prendre que l'une des deux valeurs `0` ou `1`. Ces opérateurs ne s'appliquent qu'aux types entiers : `char`, `int`, et `long`. Pour comprendre le fonctionnement de ces opérateurs, vous devez maîtriser la notation binaire, puisqu'il s'agit de la technique utilisée par l'ordinateur pour enregistrer ces entiers. Cette notation est détaillée en Annexe C.

L'utilisation la plus fréquente des opérateurs bit à bit consiste à faire dialoguer directement le programme C avec la machine. Ce sujet n'est pas traité dans ce chapitre, car il sort du cadre de ce livre. Nous allons présenter les autres applications possibles.

Les opérateurs de décalage

Le rôle des deux opérateurs de décalage est de déplacer les bits d'une variable entière d'un certain nombre de positions. L'opérateur (<<) décale les bits vers la gauche et l'opérateur (>>) les décale vers la droite. Voici la syntaxe utilisée :

```
x << n
x >> n
```

Chacun de ces opérateurs décale les bits de la variable `x` de `n` positions dans la direction correspondante. Lorsque le décalage s'effectue vers la droite, les `n` bits supérieurs reçoivent la valeur zéro. Lorsque ce décalage se fait vers la gauche, ce sont les `n` bits de plus bas niveau qui reçoivent la valeur zéro. Voici quelques exemples :

- La valeur binaire `00001100` (`12`, en décimal) décalée 2 fois à droite devient `00000011` (`3`, en décimal).
- La valeur binaire `00001100` (`12`, en décimal) décalée 3 fois à gauche devient `01100000` (`96`, en décimal).
- La valeur binaire `00001100` (`12`, en décimal) décalée 3 fois à droite devient `00000001` (`1`, en décimal).
- La valeur binaire `00110000` (`48`, en décimal) décalée 3 fois à gauche devient `10000000` (`128`, en décimal).

Ces opérateurs permettent dans certains cas de multiplier ou de diviser une variable entière par une puissance de 2. En décalant un entier de *n* positions vers la gauche, vous obtenez une multiplication par 2*n* à condition de ne "perdre" aucun bit significatif dans cette opération. Ce même décalage vers la droite permet d'obtenir une division entière par 2*n* puisque l'on perd la fraction décimale du résultat. Si vous décalez d'une position vers la droite, par exemple, la valeur `5` (`00000101`) pour la diviser par deux, le résultat sera `2` (`00000010`), plutôt que `2,5`. Le Listing 20.7 présente une utilisation de ces opérateurs.

Listing 20.7 : Les opérateurs de décalage

```
/* Les opérateurs de décalage. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned int y, x = 255;
    int count;

    printf("Valeur décimale\t\tdécalage à gauche\trésultat\n");

    for (count = 1; count < 8; count++)
    {
        y = x << count;
        printf("%d\t\t%d\t\t%d\n", x, count, y);
    }
    printf("\n\nValeur décimale\t\tdécalage à droite\trésultat\n");

    for (count = 1; count < 8; count++)
    {
         y = x >> count;
         printf("%d\t\t%d\t\t%d\n", x, count, y);
    }
    exit(EXIT_SUCCESS);
}
```

L'exécution de ce programme donne le résultat suivant :

```
Valeur décimale     Décalage à gauche   résultat
255                              1         254
255                              2         252
255                              3         248
255                              4         240
255                              5         224
255                              6         192
255                              7         128
Valeur décimale     Décalage à droite     résultat
255                              1           127
255                              2            63
255                              3            31
255                              4            15
255                              5             7
255                              6             3
255                              7             1
```

Les opérateurs logiques bit à bit

Le Tableau 20.1 présente les trois opérateurs logiques bit à bit qui permettent de manipuler les bits d'une donnée de type entier. Ces opérateurs semblent analogues aux opérateurs booléens étudiés précédemment, mais le résultat obtenu est différent.

Tableau 20.1 : Les opérateurs logiques bit à bit

Opérateur	*Description*
`&`	ET
`|`	OU inclusif
`^`	OU exclusif

Ces opérateurs binaires attribuent la valeur `0` ou `1` aux bits du résultat en fonction des bits constituant les opérandes. Ils fonctionnent de la façon suivante :

- L'opérateur ET bit à bit attribue la valeur `1` à un bit du résultat lorsque les deux bits correspondants des opérandes ont la valeur `1`. Dans le cas contraire, il définit le bit à `0`. Cet opérateur est utilisé pour désactiver ou remettre à zéro un ou plusieurs bits dans une valeur.
- L'opérateur OU inclusif bit à bit attribue la valeur `0` à un bit du résultat si les deux bits correspondants des opérandes ont la valeur `0`. Dans le cas contraire, il attribue la valeur `1`. Cet opérateur est utilisé pour activer ou définir un ou plusieurs bits dans une valeur.

- L'opérateur OU exclusif bit à bit attribue la valeur `1` à un bit du résultat si les bits correspondants des opérandes sont différents. Dans le cas contraire, il attribue la valeur `0`.

Voici quelques exemples mettant en œuvre ces opérateurs :

Opération	***Exemple***

ET	`  11110000`
	`& 01010101`
	
	`  01010000`

OU inclusif	`  11110000`
	`| 01010101`
	
	`  11110101`

OU exclusif	`  11110000`
	`^ 01010101`
	
	`  10100101`

Voici pourquoi on peut utiliser le ET bit à bit et le OU inclusif bit à bit pour remettre à zéro et définir, respectivement, certains bits d'une valeur entière. Supposons que vous vouliez remettre à zéro les bits en positions 0 et 4 d'une variable de type `char`, tout en conservant la valeur initiale des autres bits. Vous obtiendrez ce résultat en combinant cette variable avec la valeur binaire `11101110` à l'aide de l'opérateur ET.

Pour chaque valeur `1` du second opérande, le résultat sera égal à la valeur correspondante dans le premier opérande :

```
0 & 1 == 0
1 & 1 == 1
```

Pour chaque valeur `0` du second opérande, le résultat sera égal à `0` quelle que soit la valeur correspondante dans le premier opérande :

```
0 & 0 == 0
1 & 0 == 0
```

L'opérateur OU opère de façon similaire. Pour chaque valeur `1` du second opérande, le résultat sera égal à `1` et pour chaque valeur `0` du second opérande, la valeur correspondante dans le premier opérande restera inchangée :

```
0 | 1 == 1
1 | 1 == 1
0 | 0 == 0
1 | 0 == 1
```

L'opérateur complément

L'opérateur unaire complément (~) est le dernier opérateur bit à bit. Sa fonction consiste à inverser tous les bits de son opérande. `254` (`11111110`), par exemple, va se transformer en `1` (`00000001`).

Tous les exemples de cette section font intervenir des variables de type `char` constituées de 8 bits. Le fonctionnement de ces opérations est identique dans le cas de variables plus longues comme les types `int` et `long`.

Les champs de bits dans les structures

Nous allons terminer cette étude avec les champs de bits des structures. Vous avez appris au Chapitre 11 à définir vos propres structures de données et à les adapter aux besoins de votre programme. Les champs de bits permettent de personnaliser encore davantage ces données et de réduire la mémoire nécessaire.

Un *champ de bits* est un membre de structure constitué d'un certain nombre de bits. Vous déclarez ce champ en indiquant le nombre de bits nécessaires pour recevoir les données.

Supposons que vous créez une base de données des employés pour votre entreprise. Cette base de données va contenir de nombreux éléments d'information du type oui/non pour indiquer si l'employé est diplômé de l'université, par exemple, ou s'il participe au plan de prévention dentaire. Chacune de ces informations peut être enregistrée en un seul bit, la valeur `1` indiquant une réponse positive et la valeur `0`, une réponse négative.

La plus petite entité utilisée dans une structure avec les types de données standards du C est le type `char`. Vous pouvez bien sûr faire appel à ce type dans votre membre de structure pour enregistrer vos données oui/non, mais sept bits sur les huit qui constituent la variable `char` seront inutilisés. Les champs de bits permettent d'enregistrer huit réponses oui/non dans une seule variable `char`.

Les valeurs oui/non ne sont pas les seules applications des champs de bits. Imaginons que l'entreprise de notre exemple offre trois possibilités pour une assurance complémentaire maladie. Votre base de données devra enregistrer l'assurance choisie par chaque employé. La valeur `0` pourrait signifier aucune assurance souscrite, et les valeurs `1`, `2`, et `3` pourrait représenter la souscription à l'une de ces assurances. Un champ de bits constitué de 2 bits sera suffisant pour enregistrer les quatre valeurs de `0` à `3`. Un champ de bits sur trois positions pourra de la même façon recevoir des valeurs entre `0` et `7`, quatre bits pourront enregistrer des valeurs comprises entre `0` et `15`, etc.

On accède aux champs de bits comme à un membre de structure ordinaire. Ils sont tous du type `unsigned int` et leur taille est indiquée (en bits) après deux points, à la suite du nom du

membre. Les lignes qui suivent définissent une structure constituée d'un membre `universite` sur un bit, et d'un membre `sante` de 2 bits :

```
struct emp_data {
unsigned universite     : 1;
unsigned sante          : 2;
...
};
```

Les trois points indiquent la présence éventuelle d'autres membres dans cette structure, de type champ de bits, ou constitués d'un type de donnée standard. Notez que les champs de bits doivent apparaître en tête de la définition de la structure, et que l'on accède à ce type de membre de façon standard. La définition de structure précédente peut être complétée comme suit :

```
struct emp_data {
unsigned universite     : 1;
unsigned sante      : 2;
char fname[20];
char lname[20];
char ssnumber[10];
};
```

Vous pouvez ensuite déclarer le tableau de structures suivant :

```
struct emp_data workers[100];
```

Voici comment attribuer des valeurs au premier élément du tableau :

```
workers[0].universite = 0;
workers[0].sante = 2;
strcpy(workers[0].fname, "Mildred");
```

Vous pouvez bien sûr simplifier votre code en utilisant les constantes symboliques `OUI` et `NON` avec des valeurs de `0` et `1` lorsque vous travaillez avec des champs d'un bit. Vous devez considérer chaque champ de bits comme un entier non signé constitué d'un nombre donné de bits. On pourra attribuer à chacun de ces champs des valeurs entre `0` et `2n 1`, `n` étant le nombre de bits du champ. Si vous attribuez une valeur n'appartenant pas à cet intervalle, le compilateur ne signalera pas votre erreur, et vous obtiendrez des résultats erronés.



À faire

Utiliser des constantes définies `OUI` et `NON` ou `VRAI` et `FAUX` lorsque vous travaillez au niveau des bits. Le code sera plus facile à relire qu'en utilisant des `0` et des `1`.

À ne pas faire

Définir des champs avec 8 ou 16 bits. Utilisez plutôt une variable équivalente du type `char` *ou* `int`*.*

Résumé

Dans ce chapitre, nous avons traité plusieurs sujets : les conversions de types, l'allocation de mémoire dynamique et les fonctions opérant directement sur la mémoire : initialisation et copie. Vous avez aussi vu comment et quand utiliser la coercition sur les variables et les pointeurs. Le mauvais usage de la coercition est une des causes d'erreur les plus fréquentes en C. Vous avez enfin étudié les différentes méthodes de manipulation au niveau des bits.

Q & R

Q Quel est l'avantage de l'allocation de mémoire dynamique ? Pourquoi est-ce que je ne peux pas tout simplement déclarer la place mémoire dont j'ai besoin dans mon programme source ?

R Parce que vous ne la connaissez pas toujours. La place nécessaire peut dépendre des données que vous allez traiter.

Q Pourquoi dois-je toujours libérer la mémoire acquise dynamiquement ?

R Plus vos programmes se rapprocheront de la réalité, plus ils grossiront et plus vous aurez besoin de mémoire. Ce n'est pas une denrée inépuisable et il faut apprendre à la gérer. C'est particulièrement vrai dans un environnement multitâche comme Windows ou Linux.

Q Qu'arrivera-t-il si je réutilise une chaîne de caractères sans appeler `realloc()` ?

R Si vous ne risquez pas de dépasser la place allouée pour votre chaîne, vous n'avez pas besoin d'appeler `realloc()`. N'oubliez pas que C est un langage permissif, qui vous autorise donc à faire des choses que vous ne devriez, raisonnablement, jamais faire. Si vous écrasez une chaîne par une autre, plus grande, vous allez déborder de la place allouée et, au mieux, faire arrêter votre programme sur une erreur de segmentation et, au pire, piétiner autre chose : variable, programme. C'est pour éviter ce genre de problèmes que vous devez préalablement appeler `realloc()`.

Q Quel est l'avantage de la famille `mem...()` ? Pourquoi ne pas utiliser tout simplement une boucle `for` ?

R Ce n'est pas interdit, mais les fonctions dont vous parlez font la même chose plus rapidement, avec, cependant, des limitations (surtout pour `memset()`).

Q Quelles sont les applications des opérateurs de décalage et des opérateurs logiques bit à bit ?

R Ces opérateurs sont utilisés la plupart du temps lorsque le programme dialogue directement avec la machine. Ce type d'opération nécessite souvent la génération et l'interprétation d'un modèle spécifique de bits. Ce sujet n'est pas traité dans ce livre. Les opérateurs de décalage permettent, dans certains cas, de diviser ou de multiplier des valeurs entières par des puissances de 2.

Q Quels sont les avantages de l'utilisation des champs de bits ?

R Considérons le cas d'un sondage qui constitue un exemple analogue à celui fournit dans ce chapitre. Les utilisateurs doivent répondre aux questions posées par oui ou par non. Si vous posez cent questions à dix mille personnes et que vous enregistriez chaque réponse dans un type `char`, vous devrez disposer de 10 000 × 100 octets de mémoire (un caractère occupe en effet 1 octet). Cela représente un million d'octets. Si vous optez, dans ce cas, pour les champs de bits, vous pourrez enregistrer huit réponses par octet (puisqu'un octet est constitué de 8 bits). Le besoin en mémoire se réduit ainsi à 130 000 octets.

Atelier

L'atelier vous propose quelques questions permettant de tester vos connaissances sur les sujets que nous venons d'aborder dans ce chapitre.

Quiz

1. Quelle différence y a-t-il entre `malloc()` et `calloc()` ?
2. Quelle est la raison la plus courante d'utiliser la coercition sur une variable numérique ?
3. Quel est le type du résultat obtenu par l'évaluation des expressions suivantes, sachant que `c` est de type `char` ; `i`, de type `int` ; `l`, de type `long` et `f`, de type `float` ?
   a) (c + i + 1)
   b) (i + 32)
   c) (c + 'A')
   d) (i + 32.0)
   e) (100 + 1.0)
4. Que signifie l'expression "allocation de mémoire dynamique" ?
5. Quelle différence y a-t-il entre `memcpy()` et `memmove()` ?

6. Votre programme utilise une structure qui doit stocker (dans l'un de ses membres) le jour de la semaine sous la forme d'une valeur entre 1 et 7. Quelle technique faut-il choisir pour utiliser au mieux votre mémoire ?
7. Quel est, en définissant une structure, le minimum de mémoire nécessaire pour enregistrer la date courante ? (Sous la forme mois/jour/année ; considérez 1900 comme point de départ pour le compte des années.)
8. Quelle est la valeur de `10010010 << 4` ?
9. Quelle est la valeur de `10010010 >> 4` ?
10. Quelles sont les différences entre les résultats des deux expressions suivantes :

```
(01010101 ^ 11111111 )
( ~01010101 )
```

Exercices

1. Écrivez une commande `malloc()` qui alloue de la place pour 1 000 éléments de type `long`.
2. Écrivez une commande `calloc()` qui alloue de la place pour 1 000 éléments de type `long`.
3. En supposant que vous ayez déclaré ainsi un tableau :

```
float data[1000];
```

donnez deux façons d'initialiser tous ses éléments à zéro, dont l'une avec une boucle et l'autre, sans.

4. **CHERCHEZ L'ERREUR :** Y a-t-il une erreur dans les instructions ci-après ?

```
void fonc()
{ int nombre1=100, nombre2=3;
  float reponse;

  reponse = nombre1 / nombre2;
  printf("%d/%d = %lf\n", nombre1, nombre2, reponse);
}
```

5. **CHERCHEZ L'ERREUR :** Y a-t-il une erreur dans les instructions ci-après ?

```
void *p;
p = (float*) malloc(sizeof(float));
*p = 1.23;
```

6. **CHERCHEZ L'ERREUR :** La structure suivante est-elle correcte ?

```
struct quiz_answers {
char student_name[15];
unsigned answer1   : 1;
unsigned answer2   : 1;
unsigned answer3   : 1;
unsigned answer4   : 1;
unsigned answer5   : 1;
}
```

Les exercices qui suivent ne sont pas corrigés en Annexe G.

7. Créez un programme qui fait appel à tous les opérateurs logiques bit à bit. L'opérateur doit être appliqué à un nombre, puis de nouveau au résultat obtenu. Étudiez la sortie du programme afin de bien comprendre le processus.
8. Créez un programme qui affiche la valeur binaire d'un nombre (utilisez pour cela les opérateurs bit à bit).

21

Utilisation avancée du compilateur

Dans ce chapitre, nous allons étudier quelques fonctionnalités supplémentaires du compilateur C :

- Utilisation de plusieurs fichiers sources
- Emploi du préprocesseur
- Exploitation des arguments de la ligne de commande

Utilisation de plusieurs fichiers sources

Jusqu'ici, tous vos programmes C étaient constitués d'un seul et unique fichier source. Pour de simples petits programmes, c'était bien suffisant. Mais rien n'empêche de diviser le fichier source en plusieurs fichiers. C'est ce qu'on appelle la *programmation modulaire*. Quel intérêt y a-t-il à procéder ainsi ?

Avantages de la programmation modulaire

La principale raison d'utiliser la programmation modulaire est liée de très près à la programmation structurée et à une forme d'écriture faisant un usage intensif des fonctions. Au fur et à mesure que vous acquerrez de l'expérience, vous serez amené à créer des fonctions d'intérêt général que vous pourrez utiliser, non seulement dans le programme pour lequel vous les avez écrites à l'origine, mais aussi dans d'autres programmes. Par exemple, vous pourriez écrire une collection de fonctions destinées à afficher des informations sur l'écran. En conservant ces fonctions dans un fichier séparé, il vous sera facile de les réemployer dans différents programmes devant faire des affichages sur l'écran. Lorsque vous écrivez un programme faisant appel à plusieurs fichiers de code source, chaque fichier est appelé un *module*.

Techniques de programmation modulaire

Un programme C ne peut avoir qu'une seule fonction `main()`. Le module qui contient cette fonction est appelé le *module principal* et les autres, les *modules secondaires*. On associe généralement un fichier d'en-tête séparé à chaque module secondaire, comme nous allons bientôt le voir. Pour l'instant, considérons quelques exemples simples illustrant les bases de la programmation modulaire. Les Listing 21.1, 21.2, et 21.3 vous montrent respectivement le module principal, le module secondaire et le fichier d'en-tête d'un programme qui lit un nombre donné par l'utilisateur et affiche son carré.

Listing 21.1 : list21_1.c : le module principal

```
/* Entrer un nombre et afficher son carré. */
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"

int main()
{
   int x;

   printf("Tapez un nombre entier : ");
   scanf("%d", &x);
```

```
       printf("\nLe carré de %d est %ld.\n", x, sqr(x));
       exit(EXIT_SUCCESS);
    }
```

Listing 21.2 : calc.c : le module secondaire

```
/* Module contenant une fonction de calcul. */

#include "calc.h"

long sqr(int x)
{
   return (long)x * x;
}
```

Listing 21.3 : calc.h : le fichier d'en-tête pour calc.c

```
/* calc.h : fichier d'en-tête pour calc.c. */

long sqr(int x);

/* fin de calc.h */
```



```
Tapez un nombre entier : 525

Le carré de 525 est 275625.
```

Analyse

Regardons en détail les trois composantes de ce programme. Le fichier d'en-tête, calc.h, contient le prototype de la fonction `sqr()` appelée dans calc.c. Comme tout module appelant la fonction `sqr()` a besoin de connaître son prototype, il faut donc inclure calc.h dans calc.c.

Le module secondaire, calc.c, contient la fonction `sqr()`. On peut y voir l'inclusion de calc.h dont le nom est placé entre guillemets et non entre les signes habituels < et >. Nous en verrons la raison un peu plus loin.

Le module principal, list21_1.C, contient la fonction `main()`. Il contient aussi un `#include` du fichier d'en-tête calc.h.

Une fois que vous avez créé ces fichiers, comment allez-vous les associer pour les compiler et confectionner le module exécutable ? C'est au compilateur que va revenir cette tâche. Si vous utilisez une ligne de commande, vous écrirez, par exemple :

```
xxx list21_1.c calc.c -o list21_1
```

où `xxx` représente la commande de votre compilateur, à priori gcc ou cc.

Avec des environnements intégrés, vous utiliserez généralement un menu. Le manuel de référence ou l'aide en ligne du compilateur utilisé vous indiquera le détail du processus à suivre.

De la sorte, le compilateur va produire les modules list21_1.o ET calc.o (list21_1.obj et calc.obj, sous certains systèmes) et l'éditeur de liens va les lier, en compagnie des fonctions de bibliothèque nécessaires, pour fabriquer list21_1 (ou list21_1.exe sur Windows). Dans certains cas, comme avec la ligne de commande ci-dessus, vous ne verrez pas les fichiers correspondant aux modules.

Composantes des modules

Comme vous le voyez, tout cela reste très simple. La seule question qui se pose réellement est de savoir ce que chaque module doit contenir. Nous allons vous donner quelques indications générales.

Le module secondaire devrait renfermer les fonctions utilitaires, celles qui sont réutilisables dans d'autres programmes. Certains programmeurs créent un module par fonction, ce qui est sans doute excessif. Mieux vaut créer un module par type de fonction. Mettez, par exemple, dans un même module, les fonctions qui ont trait au clavier, dans un autre, celles qui concernent l'écran, dans un troisième, celles qui font certaines manipulations particulières des chaînes de caractères, et ainsi de suite.

Le procédé que nous venons de voir pour compiler les modules isolés est généralisable à plus de deux modules. Peu importe l'ordre dans lequel vous allez écrire la liste de vos modules.

En ce qui concerne les fichiers d'en-tête, il faut, là encore, se garder d'en multiplier le nombre. Les programmeurs en écrivent généralement autant de modules, à raison d'un par module, dans lesquels ils font figurer le prototype des fonctions du module correspondant.

En général, on évite de mettre des instructions exécutables dans un fichier d'en-tête. On y trouve principalement des prototypes de fonctions et des `#define` (définissant les constantes symboliques et les macros).

Comme un même fichier d'en-tête peut être inclus dans plusieurs modules source, il faut éviter que certaines portions ne soient compilées plusieurs fois. Cela se fait à l'aide de directives conditionnelles que nous étudierons plus loin, dans ce même chapitre.

Variables externes et programmation modulaire

Souvent, le seul moyen de communiquer des données entre le module principal et les modules secondaires est d'échanger des arguments et des valeurs de retour avec des fonctions. Il n'y a, dans ce cas, aucune précaution particulière à prendre en ce qui concerne la

visibilité des variables. Mais il est parfois nécessaire qu'une ou plusieurs variables puissent être vues (partagées) par plusieurs modules différents.

Au Chapitre 12, nous avons dit qu'une variable externe était une variable déclarée en dehors de toute fonction. Une telle variable est visible dans la totalité du fichier source où elle est déclarée. Cependant, elle n'est visible que de l'intérieur du module où elle se trouve. Si plusieurs modules sont présents et qu'ils doivent pouvoir utiliser cette variable, il est nécessaire de la déclarer à l'aide du mot clef `extern`. Si, par exemple, vous voulez qu'une variable `taux d interet` soit visible par tous les modules, vous allez la déclarer dans l'un d'entre eux de la façon suivante :

```
float taux_d_interet;
```

en dehors de toute fonction. Dans les modules qui doivent partager cette variable, vous écrirez :

```
extern float taux_d_interet;
```

Figure 21.1
Utilisation du mot clé extern pour rendre une variable visible par plusieurs modules.

```
/* module principal */
int x, y;
int main()
{
...
...
}
```

```
/* module secondaire mod2.c */
extern int x;
fonc2()
{
...
}
...
```

```
/* module secondaire mod1.c */
extern int x, y;
fonc1()
{
...
}
...
```

Ce mot indique au compilateur qu'il ne doit pas réserver de place en mémoire pour cette variable. C'est à l'édition de liens que l'adressage de cet avant-plan sera résolu. La Figure 21.1 illustre ce mécanisme.

Dans la Figure 21.1, la variable `x` est visible dans les trois modules, alors que la variable `y` n'est visible que dans le module principal et dans le module secondaire mod1.c.

Utilisation des fichiers .o

Une fois que vous avez écrit et mis au point un module secondaire, vous pouvez le compiler et générer un fichier objet. Ainsi, il n'est plus nécessaire de le recompiler chaque fois que vous éditez un autre module de votre programme Le temps de compilation ainsi gagné

est d’ailleurs un autre avantage de la programmation modulaire. Vous êtes alors en possession du module objet (fichier .obj ou .o), de même nom que le module source que le compilateur a placé sur le disque dur. Pour compiler un objet à partir des sources d’un module, en ligne de commande, vous devez utiliser l’option -c si votre compilateur est cc ou gcc. Par exemple, les objets main.o calc.o sont obtenus ainsi :

```
gcc -c main.c
gcc -c calc.c
```

Cette ligne illustre ce qu’est la compilation. Lorsque vous avez compilé tous vos modules (et obtenu autant de fichiers .o que de modules), vous passez à l’édition des liens, qui consiste à rassembler tous ces objets en un seul de manière à obtenir l’exécutable. Si sur Unix (et Linux) l’éditeur de lien s’appelle ld, vous pouvez l’invoquer à partir du compilateur (cc ou gcc). Vous indiquez alors sur la même ligne de commande tous les objets et ajoutez l’option -o pour spécifier le nom de l’exécutable à générer. Reprenons notre exemple :

```
gcc main.o calc.o -o list21_1
```

Vous pouvez obtenir le même résultat dans un environnement intégré. Le manuel de référence de votre compilateur vous donnera toutes indications utiles sur ce point.



À faire

Créer des fonctions génériques intelligemment regroupées par types, dans plusieurs fichiers sources.

À ne pas faire

Associer plusieurs modules dans lesquels se trouveraient plusieurs fonctions `main()`.

L’utilitaire *make*

En dehors de très petits programmes, on n’utilise généralement pas une ligne de commande, mais un fichier Makefile dans lequel on va décrire l’association des différents modules du projet, leur compilation, leur édition de liens et les bibliothèques qui devront être utilisées. L’écriture correcte d’un tel module peut devenir très complexe et sort nettement des objectifs de ce livre. Nous nous contenterons de donner ici quelques indications générales.

Le principe d’un `make`, c’est de définir les *dépendances* qui existent entre les modules. Imaginons un projet qui associerait un programme principal, program.c et un module secondaire, second.c. Il existerait aussi deux fichiers d’en-tête : program.h et second.h, appelés dans program.c. Seul, second.h serait appelé dans second.c. Dans program.c seraient appelées des fonctions présentes dans second.c.

program.c est dit dépendant de deux fichiers d'en-tête parce qu'il contient un `#include` de chacun d'eux. Si vous apportez une modification à l'un des deux fichiers d'en-tête, vous devrez recompiler program.c. Mais, si cette modification ne concernait que program.h, il ne sera pas nécessaire de recompiler aussi second.c puisqu'il ne l'utilise pas. second.c ne dépend que de second.h.

L'utilitaire `make` (parfois appelé `nmake`) va "deviner" les relations de dépendance d'après les dates des différents fichiers et prendre ses décisions de recompilation sur cette base. Pour vous donner une idée de ce que serait alors le fichier Makefile, nous vous le donnons ici sans explication. Faites attention : les lignes qui semblent commencer par des espaces commencent en réalité par une tabulation, ce qui est obligatoire dans la syntaxe d'un fichier Makefile.

```
program: program.o second.o
	gcc program.o second.o -o program

.c.o:
	gcc -o $@ -c $<
```

Le préprocesseur C

Le préprocesseur fait partie intégrante du compilateur proprement dit. Lorsque vous compilez un programme C, c'est le préprocesseur qui (comme son nom le laisse deviner) va s'attaquer en premier à votre fichier source. Une fois son travail fait, il va, de lui-même, appeler le compilateur. Selon les éditeurs de compilateurs, le préprocesseur peut ou non être un module séparé de celui du compilateur. Cela ne change rien à son *modus operandi*.

Le préprocesseur est directement concerné par les *directives* qui figurent dans vos modules source. Il les décode et c'est le résultat de ce traitement qui va être soumis au compilateur. Normalement, vous ne voyez jamais ce fichier intermédiaire qui est automatiquement supprimé par le compilateur, une fois qu'il l'a utilisé. Nous verrons, cependant, qu'il est possible de voir ce qu'il contient.

Nous allons commencer par examiner les directives traitées par le préprocesseur. Elles commencent toutes par le caractère dièse (`#`).

La directive *#define*

Cette directive sert à deux fins : définir des constantes symboliques et créer des macros.

Macros de substitution

Au Chapitre 3, vous avez appris les bases de l'utilisation des macros de substitution pour la création de constantes symboliques. Leur forme générale est :

```
#define texte1 texte2
```

Cette directive dit au préprocesseur de remplacer toutes les occurrences de `texte1` dans le programme par `texte2`, sauf lorsque `texte1` est placé entre guillemets.

L'usage le plus fréquent de cette directive est donc de créer des constantes symboliques. Si, par exemple, votre programme contient les lignes suivantes :

```
#define MAX 1000

x = y * MAX;
z = MAX - 12;
```

Le code source est transformé en :

```
x = y * 1000;
z = 1000 - 12;
```

L'effet produit est identique à celui que vous auriez obtenu en utilisant la fonction de remplacement de votre traitement de texte. Le code source lui-même reste inchangé, c'est la copie intermédiaire qui sera passée au compilateur qui garde trace de ces transformations.

Notez que cette substitution n'est pas limitée à des noms de variables ou de constantes et peut s'appliquer à n'importe quelle chaîne de caractères du fichier source. Par exemple :

```
#define ONVAVOIR printf

ONVAVOIR("Hello, world");
```

Mais, en dehors de la production de codes ésotériques et difficile à relire, cette particularité est rarement utilisée.

Création de macros avec *#define*

Vous pouvez aussi utiliser `#define` pour créer des macros de type fonction qui sont, en quelque sorte, des notations abrégées destinées à représenter quelque chose de plus compliqué. On les appelle parfois des "fonctions macro" parce que, tout comme les fonctions, elles acceptent des arguments. Ces arguments ne sont pas typés.

Prenons un exemple. Considérons la directive :

```
#define MOITIE(valeur) ((valeur)/2)
```

Elle définit une macro appelée `MOITIE` qui accepte un argument appelé `valeur`. Lorsque le préprocesseur rencontre la chaîne `MOITIE(...)` dans le texte du fichier source (... représentant n'importe quoi), il remplace l'ensemble par le texte de définition en reproduisant l'argument passé. Exemple :

```
resultat = MOITIE(10);
```

devient :

```
resultat = ((10)/2);
```

De la même façon :

```
printf("%f\n", MOITIE(x[1] +  y[2]));
```

deviendra :

```
printf("%f\n", ((x[1] +  y[2])/2));
```

Une macro peut accepter plusieurs arguments, chacun d'entre eux pouvant être utilisé plusieurs fois dans le texte de remplacement. Par exemple, la macro suivante, qui calcule la moyenne de cinq valeurs, accepte cinq arguments :

```
#define MOYENNE(u, v, w, x, y) (((u)+(v)+(w)+(x)+(y))/5)
```

Dans cette autre macro où intervient l'opérateur ternaire conditionnel, on détermine la plus grande de deux valeurs. Elle utilise chaque argument deux fois (nous avons étudié l'opérateur conditionnel au Chapitre 4).

```
#define PLUSGRAND(x,y) ((x)>(y)?(x):(y))
```

Tous les arguments figurant dans la liste de la macro ne doivent pas obligatoirement être utilisés dans le texte de remplacement. Ainsi, dans cet exemple, il n'y a rien d'illégal :

```
#define ADD(x, y, z) ((x)+(y))
```

En revanche, il faudra appeler `ADD` avec trois arguments dont le troisième ne servira à rien, sinon à se conformer à la définition de la macro. On voit, une fois de plus, que C n'interdit pas d'écrire des bêtises !

L'emploi des parenthèses est plus sévèrement réglementé qu'à l'intérieur des instructions "normales". En particulier, la première parenthèse ouvrante doit être accolée au nom de la macro. C'est de cette façon que le préprocesseur sait qu'il s'agit d'une macro et non d'une simple définition de constante. Dans l'exemple ci-avant, écrire :

```
#define ADD (x, y, z) ((x)+(y))
```

reviendrait à déclarer une substitution généralisée de la chaîne `ADD` par la chaîne `(x, y, z) ((x)+(y))`, ce qui n'aurait pas du tout l'effet escompté.

Il ne faut pas croire que les parenthèses entourant chaque nom d'argument soient une coquetterie. Elles sont indispensables pour éviter des effets de bord parasites, parfois difficilement décelables. Considérons l'exemple simple suivant :

```
#define CARRE(x) x*x
```

Si on appelle CARRE avec un argument simple, comme `x` ou `3.14`, il n'y aura pas de problème. Mais, que va-t-il se passer si on écrit :

```
z = CARRE(x + y);
```

Le préprocesseur va transformer cette instruction en :

```
z = x + y * x + y;
```

Ce n'est pas du tout ce qu'on voulait faire. Alors que si on avait entouré chaque argument d'une parenthèse :

```
z = CARRE((x) +  (y));
```

on aurait obtenu :

```
z = (x + y) * (x + y);
```

qui est sans doute plus conforme à ce qu'on espérait.

On peut apporter davantage de souplesse à l 'utilisation des macros grâce à l'opérateur # précédant immédiatement le nom d'un argument. Celui-ci est alors transformé en chaîne de caractères lors de l'expansion de la macro. Si on écrit :

```
#define SORTIE(x) printf(#x)
```

et qu'on utilise cette macro comme ceci :

```
SORTIE(Salut, les copains!);
```

on obtiendra :

```
printf("Salut, les copains !");
```

Cette "caractérisation" prend en compte tous les caractères, mêmes ceux qui ont un sens particulier et ceux qui demandent un caractère d'échappement. Dans l'exemple ci-avant, si on avait écrit :

```
SORTIE("Salut, les copains !");
```

on aurait obtenu la substitution :

```
printf("\"Salut, les copains !\"");
```

L’exemple du Listing 21.4 vous présente une application de l’opérateur (#). Mais, avant de vous y attaquer, il faut que nous étudions un autre opérateur, celui de *concaténation* (##). Cet opérateur concatène (joint) deux chaînes de caractères dans l’expansion d’une macro. Il ne traite pas les caractères d’échappement. Son utilisation principale est de créer des suites de codes sources. Si, par exemple, vous définissez une macro :

```
#define CHOP(x) fonc ## x
salade = CHOP(3)(q, w);
```

il en résultera l’expansion :

```
salade = fonc3 (q, w);
```

Vous constatez qu’ainsi, vous pouvez modifier le nom de la fonction appelée, donc, en définitive, le code source.

Listing 21.4 : Utilisation de l’opérateur (#) dans une expansion de macro

```
 1:   /* Illustre l'utilisation de l'opérateur #
 2:      dans l'expansion d'une macro. */
 3:   #include <stdio.h>
 4:   #include <stdlib.h>
 5:
 6:   #define OUT(x) printf(#x " est égal à %d.\n", x)
 7:
 8:   int main()
 9:   {
10:      int valeur = 123;
11:
12:      OUT(valeur);
13:      exit(EXIT_SUCCESS);
14:   }
```



```
valeur est égal à 123.
```

Analyse

L’opérateur (#) de la ligne 6 permet de reporter tel quel `le nom` de la variable passée en argument, sous forme de chaîne de caractères, dans l’expansion de la macro qui devient :

```
printf("valeur" " est égale à %d.\n", valeur);
```

Macros ou fonctions ?

Vous venez de voir que les macros pouvaient être utilisées aux lieu et place de véritables fonctions, tout au moins dans des situations où le code résultant est relativement court. Les macros peuvent dépasser une ligne mais, généralement, elles deviennent trop difficiles à maîtriser en quelques lignes. Lorsque vous avez le choix entre une fonction ou une macro, laquelle devez-vous choisir ? C'est une question de compromis entre la taille et la vitesse d'exécution du programme.

Une définition de macro voit son expansion directement insérée dans le code généré par le compilateur chaque fois qu'elle est appelée. Si vous avez 100 appels de la macro, son expansion sera insérée 100 fois dans votre programme. Au contraire, le code d'une fonction n'existe qu'en un seul exemplaire. En ce qui concerne l'encombrement, la palme revient donc à la fonction.

En revanche, à chaque appel de fonction est associé un certain *overhead* (surcharge) de temps CPU, causé par le mécanisme de liaison et de passage des arguments d'une part et par le renvoi du résultat d'autre part. Ce n'est pas le cas pour une macro, puisqu'il n'y a pas d'appel, son expansion étant directement insérée dans le code. Ici, c'est donc la macro qui est sur la plus haute marche du podium.

Pour le programmeur débutant, ces considérations sont, en général, peu importantes. Elles ne deviennent préoccupantes que lorsque l'on s'attaque à de gros programmes ou à des programmes dont la vitesse d'exécution ou l'encombrement en mémoire est crucial.

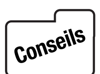
Comment examiner l'expansion d'une macro

Il y a des moments où vous aimeriez pouvoir contempler ce que le préprocesseur a fait lors de l'expansion d'une macro, ne serait-ce que pour comprendre pourquoi elle ne se comporte pas comme vous l'espériez. Pour cela, il faut demander au compilateur de créer un fichier contenant le résultat du traitement par le préprocesseur ou appeler directement celui-ci. Cela dépend du compilateur que vous utilisez. En mode ligne de commande, cependant, on peut appeler le préprocesseur de la façon suivante :

```
cpp program.c
```

Selon le préprocesseur, le résultat pourra être affiché directement à l'écran ou mis dans un fichier ayant l'extension .i et dans lequel vous trouverez l'expansion de votre code précédée de celle des fichiers d'include, ce qui peut conduire à un fichier assez gros. C'est vers la fin que se trouve ce qui découle directement de vos propres instructions.

Il ne vous reste plus qu'à charger le fichier obtenu dans un éditeur de texte pour l'examiner à loisir.



À faire

Utiliser `#define` pour définir des constantes symboliques qui rendent les programmes plus faciles à lire et à maintenir. Vous pouvez ainsi définir des couleurs, les mots OUI, NON, VRAI et FAUX, (pour un test, par exemple), TOUJOURS et JAMAIS (pour un `while`, par exemple).

À ne pas faire

Abuser des macros "fonctions". Tant que vous n'aurez pas une bonne expérience du C, vous risquez d'avoir des surprises !

La directive #*include*

Dans tous les chapitres qui précèdent, vous avez fait usage de la directive `#include` pour inclure des fichiers d'en-tête au début de votre programme. Lorsqu'il rencontre cette directive, le préprocesseur lit le fichier spécifié et l'insère dans un fichier intermédiaire (celui qu'il passera plus tard au compilateur) à l'emplacement où se trouvait l'include. Il n'est pas possible d'utiliser des caractères de remplacement (`*` ou ?) dans un nom de fichier d'include. D'ailleurs, cela n'aurait aucun sens. Vous pouvez imbriquer des inclusions de fichiers. Rien n'empêche un fichier "inclus" de contenir lui-même un `#include` d'un fichier qui, à son tour, contiendrait...

Il y a deux façons de spécifier le nom du fichier à inclure. Vous le placez soit entre les caractères < et >, soit entre guillemets. Ce choix n'est pas indifférent. Dans le premier cas, le préprocesseur va rechercher le fichier à inclure dans les répertoires standard des fichiers d'include. S'il ne le trouve pas, il consultera le répertoire courant.

"Qu'est-ce que les répertoires standard ?" vous demandez-vous peut-être. Il s'agit d'une liste de répertoires par défaut qui dépend de votre système (généralement /usr/include sur un système Unix ou Linux) et qui peut être étendue si vous utilisez l'option -I d'un compilateur en ligne de commande comme cc ou gcc. Par exemple, si vous compilez en indiquant `I/usr/local/include` à votre compilateur gcc, les répertoires standard seront /usr/include et /usr/local/include.

La seconde méthode pour spécifier un fichier d'include est de placer son nom entre guillemets comme dans `#include "monfic.h"`. Dans ce cas, le préprocesseur ira chercher le fichier dans le répertoire contenant le fichier source à compiler puis dans les répertoires standard. En règle générale, les fichiers d'en-tête que vous avez écrit vous-même doivent être placés dans le même répertoire que les fichiers sources. Les répertoires standard sont réservés pour les fichiers d'en-tête de bibliothèques.

#if , *#elif*, *#else* et *#endif*

Ces quatre directives gouvernent ce qu'on appelle une *compilation conditionnelle*. Cette expression signifie que certains blocs de programme ne seront compilés que si une certaine condition est remplie. La directive `#if` et ses sœurs ressemblent aux instructions `if`, `else`, etc. Mais ces dernières contrôlent l'exécution du programme alors que les directives en contrôlent la compilation.

La structure d'un bloc `#if` est la suivante :

```
#if condition_1
Bloc d'instructions 1
#elif condition_2
Bloc d'instructions 2
...
#elif condition_n
Bloc d'instructions n
#else
Bloc d'instructions par défaut
#endif
```

L'expression de test qu'utilise `#if` peut être n'importe quelle expression dont la valeur peut se réduire à une constante. L'usage de l'opérateur `sizeof()`, de la coercition ou du type `float` est interdit. En général, on utilise des constantes symboliques créées au moyen de la directive `#define`.

Chaque *Bloc d'instructions* consiste en une ou plusieurs instructions C de n'importe quel type, y compris des directives du préprocesseur. Elles n'ont pas besoin d'être imbriquées entre des accolades, mais ce n'est pas défendu.

Les directives `#if` et `#endif` sont nécessaires, mais `#else` et `#elif` sont facultatives. Vous pouvez placer autant de directives `#elif` que vous le voulez, mais une seule `#else`. Lorsque le compilateur atteint un `#if` il teste la condition associée. Si cette condition a la valeur VRAI (non zéro), les instructions qui suivent sont compilées. Si elle a la valeur FAUX (zéro), le compilateur teste, dans l'ordre, les conditions associées à chacune des `#elif` qui suivent. Les instructions qui suivent la première directive `#elif` dont la valeur testée est VRAI sont compilées. Si aucune des conditions ne vaut VRAI, les instructions suivant la directive `#else` sont compilées.

Au plus, un seul bloc d'instructions encadré entre un `#if` et un `#endif` est compilé. Si le compilateur ne trouve pas de directive `#else`, aucune instruction ne sera compilée.

L'emploi de ce mécanisme de compilation conditionnelle n'est limité que par votre imagination. En voici un exemple. Supposez que vous écriviez un programme utilisant des données dépendant d'un système d'exploitation (à cause de fonctions non portables, par exemple), vous pouvez utiliser une batterie de `#if...#endif` pour opérer une sélection parmi plusieurs fichiers d'en-tête :

```
#if SOLARIS == 1
#include "solaris.h"
#elif LINUX == 1
#include "linux.h"
#elif WINDOWS == 1
#include "windows.h"
#else
#include "generic.h"
#endif
```

Utilisation de *#if...#endif* pour la mise au point

Une autre utilisation fréquente de la construction `#if...#endif` est l'inclusion d'instructions de mise au point dans le programme. Vous pouvez donner à une constante symbolique `DEBUG` la valeur 0 ou la valeur 1 et, à des endroits choisis du programme, insérer des instructions conditionnelles :

```
#if DEBUG == 1
... instructions de mise au point ...
#endif
```

Au cours de la mise au point du programme, `DEBUG` aura la valeur 1 et à la fin, on refera une compilation après lui avoir donné la valeur 0, "effaçant" ainsi les instructions de mise au point.

On peut utiliser l'opérateur `defined` pour tester si une constante symbolique a été ou non définie. Ainsi, l'expression :

```
defined(NOM)
```

prend la valeur VRAI si `NOM` a été défini (par une directive `#define`), et FAUX dans le cas contraire. Peu importe la valeur qui lui a été donnée. Il n'est même pas nécessaire de fixer une valeur, il suffit d'écrire, par exemple :

```
#define NOM
```

On peut alors réécrire le précédent exemple sous la forme :

```
#if defined(NOM)
... instructions de mise au point ...
#endif
```

On peut aussi utiliser `defined()` pour assigner une définition à un nom, seulement si ce nom n'a pas encore fait l'objet d'un `#define` :

```
#if ! defined(MACHIN)  /* si MACHIN n'a jamais été défini */
#define MACHIN 23
#endif
```

#ifdef, #ifndef

Ces deux instructions sont équivalentes à `#if defined` et à `#if ! defined`. Nous vous les donnons pour vous permettre de comprendre plus facilement certains programmes qui les utilisent. Elles sont parfois préférables car plus courtes à écrire. Néanmoins, elles sont limitées car vous ne pouvez pas tester plusieurs conditions à la fois, ni utiliser de `#elif` ou de `#else`. Nous vous présentons un exemple ci-dessous.

Comment éviter plusieurs inclusions d'un fichier d'en-tête

Lorsqu'un programme grossit ou que vous employez des fichiers d'en-tête de plus en plus nombreux, vous courez le risque d'en inclure un plusieurs fois, ce qui pourrait poser des problèmes au compilateur. Avec ce que nous venons de voir, il est facile d'éviter ce problème (voir Listing 21.5).

Listing 21.5 : Utilisation des directives du préprocesseur avec des fichiers d'en-tête

```
1:  /* prog.hprog.h - Fichier d'en-tête comportant un test
2:     destiné à empêcher plusieurs inclusions */
3:  #ifndef PROG_H
4:  /* le fichier n'a pas encore été inclus */
5:  #define PROG_H
6:
7:  /* Informations du fichier d'en-tête */
8:
9:  #endif /* fin de prog.h */
```

Analyse

À la ligne 3, on regarde si `PROG H` est défini. Ce nom a été choisi pour permettre de repérer le fichier qu'il concerne (prog.h), mais on aurait pu aussi bien choisir TOTO. S'il est défini, on ne fait rien du tout. Si `PROG H` ne l'est pas, alors on le définit (ligne 5) et on "exécute" le contenu du fichier d'en-tête (lignes 6 à 8).

La directive *#undef*

De même qu'on peut définir un nom à l'aide d'une directive `#define`, on peut annuler cette définition par une directive `#undef`. En voici un exemple :

```
#define DEBUG 1

/* Dans cette section du programme, toutes les
   occurrences de DEBUG seront remplacées par 1
   et l'expression defined(DEBUG) vaudra VRAI.
```

```
*/

#undef DEBUG

/* Dans cette section du programme, toutes les
   occurrences de DEBUG seront remplacées par 0
   et l'expression defined(DEBUG) vaudra FAUX.
*/
```

Grâce à `#define` et à `#undef`, on peut donner à un même nom une valeur changeante dans un même programme.

Macros prédéfinies

La plupart des compilateurs contiennent un certain nombre de macros prédéfinies. Les plus utilisées sont `DATE` , `TIME` , `LINE` et `FILE` . Remarquez que ces noms sont précédés et suivis par deux blancs soulignés. Cela afin de rendre peu probable l'existence de macros de même nom accidentellement définies par le programmeur. À ce sujet, sachez qu'il est déconseillé aux programmeurs de nommer leurs constantes symboliques ou macros avec des noms commençant par un (ou deux) blanc(s) souligné(s). Elles sont réservées aux constantes prédéfinies du langage C et à certaines constantes de bibliothèques standard comme libc.

Ces macros fonctionnent comme les macros de substitution que nous avons rencontrées au début de ce chapitre : le préprocesseur remplace ces noms par les chaînes de caractères appropriées : date, heure, numéro de l'instruction dans le programme (ici, ce n'est pas une chaîne, mais une valeur décimale `int`) et nom du fichier contenant cette macro. On peut ainsi facilement afficher la date à laquelle a été compilé un module ou le numéro de ligne d'une instruction ayant causé un incident.

Voici un exemple simple d'utilisation de ces macros :

```
31:
32:  printf("Programme %s : Fichier non trouvé ligne %d\n",
            __FILE__, __LINE__);
33:
```

On obtiendra un affichage de ce genre :

```
Programme toto.c : Fichier non trouvé à la ligne 32.
```

Cela peut vous paraître peu important, mais vous risquez d'en percevoir plus nettement l'intérêt quand vos programmes seront devenus assez importants.

À faire

Utiliser les macros `FILE` *et* `LINE` *pour rendre les messages d'erreur plus précis.*

Placer des parenthèses autour de la valeur passée à une macro afin d'éviter tout effet de bord fâcheux.

À ne pas faire

Oublier le `#endif` *à la suite d'un* `#if`*.*

Les arguments de la ligne de commande

Tout programme C peut accéder aux arguments qui lui ont été passés sur sa ligne de commande, c'est-à-dire aux informations qui ont été éventuellement tapées à la suite de son nom, après l'invite du système d'exploitation. On peut par exemple écrire :

```
monprog JULES 23
```

Les deux arguments, JULES et 23, peuvent être récupérés par le programme au cours de son exécution comme des arguments passés à la fonction `main()`. On peut ainsi passer directement des informations au programme sans avoir besoin de demander explicitement à l'utilisateur de les taper. Il faut alors déclarer `main()` de la façon suivante :

```
int main(int argc, char *argv[]);
```

ou, ce qui revient au même :

```
int main(int argc, char **argv);
```

Le premier argument, `argc`, est un entier indiquant le nombre des arguments qui ont été écrits à la suite de l'invite du système d'exploitation, y compris le nom du programme lui-même. Sa plus petite valeur est donc 1. `argv[]` est un tableau de pointeurs vers des chaînes de caractères. La valeur des indices pouvant être utilisés est comprise entre 0 et `argc 1`. `argv[0]` pointe sur le nom du programme, `argv[1]`, sur le premier argument, et ainsi de suite. Les noms `argc` et `argv` ne sont pas des mots réservés, mais l'usage — très généralement respecté — veut qu'on appelle ainsi les deux arguments de `main()`.

Les arguments sont séparés les uns des autres sur la ligne de commande par des espaces. Si un des arguments que vous voulez passer est une chaîne de caractères contenant un ou plusieurs blancs, vous devez la placer entre guillemets. Comme ceci, par exemple :

```
monprog "Jules et Jim"
```

Le programme du Listing 21.6 vous donne un exemple concret d'utilisation des arguments de la ligne de commande.

Listing 21.6 : Comment récupérer les arguments passés sur la ligne de commande

```
 1:  /* Comment accéder aux arguments de la ligne de commande. */
 2:  #include <stdio.h>
 3:  #include <stdlib.h>
 4:
 5:  int main(int argc, char *argv[])
 6:  {
 7:      int count;
 8:
 9:      printf("Le nom du programme est : %s\n", argv[0]);
10:
11:      if (argc > 1)
12:
13:         for (count = 1; count < argc; count++)
14:             printf("Argument %d: '%s'\n", count, argv[count]);
15:
16:         else
17:             printf("Il n'y a pas d'argument sur la ligne de \
18:                commande.\n");
19:         exit(EXIT_SUCCESS);
20:  }
```

Exemple d'appel :

```
list21_6 Comme "un vol de gerfauts" ...

Le nom du programme est : ./list21_6
Argument 1: 'Comme'
Argument 2: 'un vol de gerfauts'
Argument 3: '...'
```

Analyse

On notera, dans ce programme, l'absence d'accolades à la suite du `if` de la ligne 11 qui teste le nombre d'arguments. Il n'y a, en effet, qu'une seule instruction (la boucle `for` de la ligne 13). Donc, il n'est pas nécessaire (mais pas défendu, non plus) d'utiliser des accolades.

Comme on le voit, le nom du programme est accompagné de son chemin d'accès. Les mots de la ligne de commande encadrés par des guillemets apparaissent bien comme un seul argument.

Les arguments de la ligne de commande peuvent être classés en deux catégories : ceux qui sont indispensables au programme pour qu'il puisse "tourner" et ceux qui sont facultatifs, comme les indicateurs qui peuvent avoir une incidence sur le comportement du

programme (afficher ou non certains résultats, trier des valeurs en ordre ascendant ou descendant, par exemple). Dans ce dernier cas, le programme adopte généralement un mode de fonctionnement par défaut lorsque ces arguments ne sont pas donnés.



À faire

Prendre l'habitude d'appeler les arguments de la ligne de commande `argc` *et* `argv`*, conformément aux bons usages de la programmation C.*

À ne pas faire

Supposer que l'utilisateur a tapé le nombre d'arguments attendus. Vérifiez. C'est facile, grâce à `argc`*. Si ce n'est pas le cas, affichez un message pour lui signaler son erreur.*

Résumé

Dans ce chapitre, nous avons abordé quelques points de la programmation C qui étendent les possibilités du langage : la programmation modulaire (éclatement d'un programme en plusieurs fichiers sources), l'utilisation du préprocesseur pour la compilation conditionnelle et le traitement des arguments de la ligne de commande.

Q & R

Q Lorsqu'un programme se compose de plusieurs modules, comment le compilateur connaît-il leurs noms ?

R Si vous appelez le compilateur sur la ligne de commande, vous énumérez les uns à la suite des autres les noms des différents modules ou vous construisez un fichier `Makefile` pour l'utilitaire `make`. Si vous utilisez un environnement intégré, vous construisez un fichier de projet.

Q L'extension .h est-elle obligatoire pour les fichiers d'en-tête ?

R Absolument pas, mais ce serait une grave entorse aux bons usages si vous en utilisiez une autre et cela compliquerait inutilement la maintenance de vos programmes.

Q Lorsque j'inclus un fichier d'en-tête dans mon programme, puis-je indiquer un chemin d'accès ?

R Oui, mais cela n'est nécessaire que si ce fichier ne se trouve ni dans le chemin d'accès décrit dans la variable d'environnement INCLUDE, ni dans celui du module source. Ce qui montre que votre fichier ne se trouve pas dans le "bon" répertoire. Ce qui

montre encore qu'il y a probablement un problème ailleurs. Mieux vaut donc régler le problème (comme ajouter une option `Ichemin` sur la ligne de commande du compilateur) plutôt que de le cacher en indiquant le chemin d'accès.

Q Y a-t-il d'autres macros prédéfinies que celles que nous venons de voir dans ce chapitre ?

R Oui. Très souvent, les éditeurs de compilateurs en rajoutent quelques-unes qui leur sont propres et servent, par exemple, à identifier la version du compilateur. De telles macros existent également dans certaines bibliothèques, en particulier la bibliothèque standard libc.

En outre, la norme ANSI a introduit la macro `STDC` dont la valeur est 1 lorsque le compilateur est conforme à cette norme. Elle existe sur tous les compilateurs "modernes". Toutefois, cela n'est pas une garantie formelle, certains éditeurs peu scrupuleux ou un peu ignorants n'hésitant pas à surqualifier leur produit un peu à la légère.

De la même façon, si vous utilisez des fonctions qui ne sont définies que dans la norme C99, comme `strtoul()` que nous avons vue au Chapitre 17, vous devrez peut-être indiquer au compilateur que vous imposez la norme C99 en définissant la constante `ISOC99 SOURCE`.

Atelier

L'atelier vous propose quelques questions permettant de tester vos connaissances sur les sujets que nous venons d'aborder dans ce chapitre.

Quiz

1. Que signifie l'expression "programmation modulaire" ?
2. Qu'appelle-t-on "module principal", en programmation modulaire ?
3. Lorsque vous définissez une macro, pourquoi chaque argument doit-il être écrit entre parenthèses ?
4. Quels sont les avantages et inconvénients d'utiliser une macro aux lieu et place d'une fonction ?
5. Que fait l'opérateur `defined()` ?
6. Que doit-on toujours trouver à la suite d'un `#if` ?
7. Quelle extension ont les fichiers compilés avant l'édition de liens ?
8. Que fait la directive `#include` ?

9. Quelle différence y a-t-il entre ces deux directives :

```
#include <monfic.h>
#include "monfic.h"
```

10. Quelle est la signification de la macro `DATE` ?
11. Sur quoi pointe `argv[0]` ?

Exercices

La pluralité de solutions possibles nous conduit à ne pas donner de solution à ces exercices :

1. Écrivez une routine d'erreur qui reçoive un numéro d'erreur, un numéro de ligne et un nom de module et affiche un message d'erreur formaté puis mette fin au programme. Vous utiliserez les macros prédéfinies lorsque c'est possible.
2. Modifiez le précédent exercice pour expliciter le type d'erreur signalé. Pour cela, vous pourrez utiliser un tableau indexé de messages, le numéro de l'erreur correspondant à l'indice du message à afficher. Bien entendu, vous prévoirez d'afficher un second message d'erreur si le numéro de l'erreur ne permet pas de retrouver de message correspondant.
3. Écrivez un programme qui accepte deux arguments sur sa ligne de commande, représentant chacun un nom de fichier, et fasse une copie du fichier représenté par le premier argument sous le nom représenté par le second argument.

Révision de la Partie III

Nous allons faire une petite révision des sept derniers chapitres que vous venez d'étudier. Le programme qui suit a été conçu pour cela.

Info

Les numéros figurant dans la marge indiquent le chapitre où a été étudiée la notion ou l'instruction de cette ligne.

```
/* Nom du programme : revis3.c
        Le programme lit des noms et des numéros de téléphone
        et les écrit sur un fichier disque dont le nom est
        indiqué sur la ligne de commande.
*/


#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

/*** Définition de constantes ***/
#define OUI          1
#define NON          0
#define REC_LENGTH  54

/*** Définition de variables ***/
struct record
{ char fname[15+1];   /* prénom terminé par NULL*/
  char lname[20+1];   /* nom propre terminé par NULL */
  char mname[10+1];   /* second prénom */
  char phone[9+1];    /* numéro de téléphone terminé par NULL */
```

```
} rec;

/*** prototypes des fonctions ***/
int  main(int argc, char *argv[]);
void display_usage (char *filename);
int  display_menu(void);
void get_data(FILE *fp, char *progname, char *filename);
void display_report(FILE *fp);
int  continue_function(void);
int  look_up(FILE *fp);

/* --- début du programme ---*/
int main(int argc, char *argv[])
{ FILE *fp;
  int  cont = OUI;


  if(argc < 2)
  { display_usage("SEMAINE3");
    exit(EXIT_FAILURE);
  }                    /* ouverture du fichier */
  if ((fp = fopen(argv[1], "a+")) == NULL)
  { fprintf(stderr, "%s(%d)--Erreur à l'ouverture du fichier \
            %s", argv[0],__LINE__, argv[1]);
    exit(EXIT_FAILURE);
  }

   while(cont == OUI)
   { switch(display_menu())
     { case '1': get_data(fp, argv[0], argv[1]);
                 break;
       case '2': display_report(fp);
                 break;
       case '3': look_up(fp);
                 break;
       case '4': printf("\n\nMerci d'avoir utilisé ce programme.\n");
                 cont = NON;
                 break;
       default:  printf("\n\nChoix incorrect. Choisissez de 1 à 4.");
                 break;
     }
   }
   fclose(fp);       /* refermer le fichier */
   exit(EXIT_SUCCESS);
}

/***************************************************
 * display_menu()                                  *
 ***************************************************/
int display_menu(void)
{ char ch, buf[20];

  printf("\n");
  printf("\n      MENU");
  printf("\n    ========\n");
  printf("\n1.   Entrée de noms");
```

```
  printf("\n2.   Affichage liste");
  printf("\n3.   Recherche numéro");
  printf("\n4.   Quitter");
  printf("\n\nTapez votre choix ==> ");
  lire_clavier(buf, sizeof(buf));
  ch = *buf;
   return(ch);  }
/*****************************************************
 *  get_data()                                       *
 *****************************************************/
void get_data(FILE *fp, char *progname, char *filename)
{ int cont = OUI;

  while(cont == OUI)
  { printf("\n\nIndiquez ci-après les renseignements :");
    printf("\n\nPrénom : ");
    lire_clavier(rec.fname, sizeof(rec.fname));

    printf("\nSecond prénom : ");
    lire_clavier(rec.mname, sizeof(rec.mname));

    printf("\nNom de famille : ");
    lire_clavier(rec.lname, sizeof(rec.lname));

    printf("\nNuméro de téléphone sous la forme 11 22 33 44 : ");
    lire_clavier(rec.phone, sizeof(rec.phone));

    if (fseek(fp, 0, SEEK_END) == 0)
     if(fwrite(&rec, 1, sizeof(rec), fp) != sizeof(rec))
     { fprintf(stderr, "%s(%d) -- Erreur en écriture sur le \
               fichier %s", progname, __LINE__, filename);
       exit(EXIT_FAILURE);
     }
    cont = continue_function();
  }
}

/********************************************************
 * display_report() - Affiche les entrées du fichier   *
 ********************************************************/
void display_report(FILE *fp)
{ time_t rtime;
  int num_of_recs = 0;

  time(&rtime);
  fprintf(stdout, "\n\nHeure actuelle : %s", ctime(&rtime));
  fprintf(stdout, "\nAnnuaire téléphonique\n");

  if(fseek(fp, 0, SEEK_SET) == 0)
  { fread(&rec, 1, sizeof(rec), fp);
    while(!feof(fp))
    { fprintf(stdout,"\n\t%s, %s %c %s", rec.lname,
              rec.fname, rec.mname[0],rec.phone);
      num_of_recs++;
      fread(&rec, 1, sizeof(rec), fp);
    }
```

```
     fprintf(stdout,"\n\nNombre total d'enregistrements : %d",num_of_recs);
     fprintf(stdout, "\n\n* * * Affichage terminé * * *\n");
   }
   else
    fprintf(stderr, "\n\n*** ERREUR AU COURS DE L'AFFICHAGE ***\n");
}

/**********************************************************
 *  continue_function()                                   *
**********************************************************/
int continue_function(void)
{ char ch;

  do
  { printf("\n\nVoulez-vous en saisir un autre ? (O)ui/(N)on ");
    lire_clavier(buf, sizeof(buf));
     ch = *buf;
  } while(strchr("NnOo", ch) == NULL);
  if(ch == 'n' || ch == 'N') return(NON);
   else                      return(OUI);
}
/******************************************************
 *  display_usage()                                   *
 ******************************************************/
void display_usage(char *filename)
{ printf("\n\nUSAGE : %s nom de fichier", filename);
  printf("\n\n       où \"nom de fichier\" est le nom du fichier \
          annuaire\n"); }

/*******************************************************
 *   look_up()                                         *
 *******************************************************/
int look_up(FILE *fp)
{ char tmp_lname[20+1];
  int  ctr = 0;

  fprintf(stdout, "\n\nIndiquez le nom propre à rechercher : ");
  lire_clavier(tmp_lname, sizeof(tmp_lname));
  if(strlen(tmp_lname) != 0)
  { if (fseek(fp, 0, SEEK_SET) == 0)
    { fread(&rec, 1, sizeof(rec), fp);
      while(! feof(fp))
      {if (strcmp(rec.lname, tmp_lname) == 0) /* si correspondance*/
        { fprintf(stdout, "\n%s %s %s - %s", rec.fname,
                          rec.lname,rec.phone);
          ctr++;
        }
        fread(&rec, 1, sizeof(rec), fp);
      }
    }
    fprintf(stdout, "\n\n%d correspondance(s).", ctr);
  }
  else
     fprintf(stdout, "\nVous n'avez pas indiqué de nom.");
  return ctr;
}
```

Analyse

Ce programme peut sembler long, mais il exécute, en réalité, peu de tâches. Il ressemble aux programmes présentés dans les deux précédentes révisions. Ici, l’utilisateur va entrer des informations destinées à constituer un annuaire téléphonique : nom, prénoms et numéro d’appel ; les capacités de cet annuaire ne sont pas limitées, car nous utilisons un fichier sur disque.

L’utilisateur peut spécifier le nom du fichier à utiliser. `main()` commence à la ligne 36 : on voit qu’il va récupérer les arguments de la ligne de commande avec `argc` et `argv`. À la ligne 41, on vérifie qu’il y a au moins un argument sur la ligne de commande. Si ce n’est pas le cas, l’instruction de la ligne 42 appelle `display usage()` pour afficher un message d’erreur et le programme se termine à la ligne 43.

Cette fonction d’affichage se trouve aux lignes 160 à 163. On notera, à la ligne 162, l’emploi d’un caractère d’échappement pour afficher les guillemets. L’emploi d’une variable (contenant le nom du programme) à la place d’une chaîne de caractères permet de toujours afficher le véritable nom du programme, même si l’utilisateur en a modifié le nom. On aurait pu utiliser la macro `__FILE__` à la place de `filename`, mais la présentation aurait été moins lisible car, alors, on aurait affiché, en plus, le nom du disque et le chemin d’accès ou, du moins, le nom du fichier source tel qu’il a été passé au compilateur.

Lorsque l’on a vérifié qu’il existait un argument, à la ligne 45 on tente d’ouvrir en mise à jour (mode `"a+"` le fichier ayant le nom qui se trouve dans `argv[1]`. Si le fichier n’existe pas et n’a pas pu être créé ou, d’une façon générale, si l’ouverture n’a pu se faire correctement, le pointeur `fp` contiendra `NULL` et on affichera un message d’erreur explicite (lignes 46 et 47) avant d’arrêter le programme (ligne 48). En cas de réussite, `fp` contient un pointeur vers le flot qui vient d’être ouvert.

On entre alors dans une boucle `while` (ligne 51) gouvernée par la variable `cont`, initialisée à `VRAI` (ligne 38). Un `switch` appelle la fonction `display menu()` et, selon la valeur renvoyée (du type `char`), choisit parmi quatre branches possibles. On sortira de cette boucle lorsque l’utilisateur aura tapé 4, ce qui correspond à quitter le programme. Pour cela, on se contente de basculer l’indicateur `cont`, ce qui va permettre de sortir du `while`. On tombe alors sur la ligne 66 où un `fclose()` ferme le fichier avant qu’à la ligne suivante, un `return 0;` ne mette fin au programme.

Si la valeur tapée n’est pas comprise entre 1 et 4, la branche `default` du `switch` (ligne 62) rappelle l’utilisateur à l’ordre et la boucle se poursuit.

La fonction `get data()` (lignes 87 à 116) possède trois arguments, respectivement, le pointeur sur le fichier ouvert, le nom du programme et celui du fichier. On trouve tout de suite une boucle `while` de structure très semblable à celle de `main()` que nous venons de voir. La variable de boucle utilisée ici est `cont`, mais, comme dans `main()`, c’est une

variable *locale*, qui n'est donc visible que de l'intérieur de la fonction. Il n'y a donc aucun risque de mélange ou de confusion. À la ligne 107, un `fseek()` permet de se positionner sur la fin du fichier pour écrire un nouveau nom. En cas d'erreur, on ne fait rien de spécial. Un vrai programme devrait au moins afficher un message d'erreur.

L'écriture dans le fichier s'effectue à la ligne 108 et, en cas d'erreur, on affiche bien un message avant de terminer par un `exit(EXIT FAILURE)`.

La fonction `display report()` (lignes 120 à 141) appelle peu de commentaires. On notera un autre usage de `fseek()` (ligne 128), ici, pour se positionner en tête du fichier. La boucle d'affichage s'arrêtera sur la détection d'une fin de fichier (ligne 130).

`continue function()` (lignes 146 à 156) demande à l'utilisateur s'il veut saisir un autre nom. On pourra remarquer, à la ligne 153, l'utilisation de `strchr()` pour tester la réponse, que celle-ci soit en minuscules ou en majuscules.

Ce programme pourrait être amélioré en transformant certaines variables locales en variables globales. Le pointeur de fichier, par exemple, serait un bon candidat pour cette transformation. On a toujours intérêt, lorsque c'est possible, à diminuer le nombre d'arguments passés à une fonction car cela gagne du temps. En général, les variables globales sont les variables qui définissent la configuration du programme. Elles sont nécessaires dans tout le code et doivent donc y être accessibles partout. C'est tout l'intérêt des variables globales. Certains programmeurs regroupent par souci de clarté leurs variables de configuration dans une structure définie pour l'occasion.



Charte des caractères ASCII

Dec.	*Hexa*	*ASCII*	*Dec.*	*Hexa*	*ASCII*
0	00	`null`	12	0C	♀
1	01	☺	13	0D	♪
2	02	☻	14	0E	♫
3	03	♥	15	0F	☼
4	04	✦	16	10	►
5	05	♣	17	11	◄
6	06	♠	18	12	↕
7	07	•	19	13	‼
8	08	◘	20	14	¶
9	09	◦	21	15	§
10	0A	◙	22	16	–
11	0B	♂	23	17	↨

Dec.	*Hexa*	*ASCII*	*Dec.*	*Hexa*	*ASCII*
24	18	↑	50	32	2
25	19	↓	51	33	3
26	1A	→	52	34	4
27	1B	←	53	35	5
28	1C	∟	54	36	6
29	1D	↔	55	37	7
30	1E	▴	56	38	8
31	1F	▾	57	39	9
32	20	espace	58	3A	:
33	21	!	59	3B	;
34	22	"	60	3C	<
35	23	#	61	3D	=
36	24	$	62	3E	>
37	25	%	63	3F	?
38	26	&	64	40	@
39	27	'	65	41	A
40	28	(	66	42	B
41	29	)	67	43	C
42	2A	*	68	44	D
43	2B	+	69	45	E
44	2C	,	70	46	F
45	2D	-	71	47	G
46	2E	.	72	48	H
47	2F	/	73	49	I
48	30	0	74	4A	J
49	31	1	75	4B	K

Dec.	*Hexa*	*ASCII*	*Dec.*	*Hexa*	*ASCII*
76	4C	L	102	66	f
77	4D	M	103	67	g
78	4E	N	104	68	h
79	4F	O	105	69	i
80	50	P	106	6A	j
81	51	Q	107	6B	k
82	52	R	108	6C	l
83	53	S	109	6D	m
84	54	T	110	6E	n
85	55	U	111	6F	o
86	56	V	112	70	p
87	57	W	113	71	q
88	58	X	114	72	r
89	59	Y	115	73	s
90	5A	Z	116	74	t
91	5B	[	117	75	u
92	5C	\	118	76	v
93	5D	]	119	77	w
94	5E	^	120	78	x
95	5F	_	121	79	y
96	60	`	122	7A	z
97	61	a	123	7B	{
98	62	b	124	7C	¦
99	63	c	125	7D	}
100	64	d	126	7E	~
101	65	e	127	7F	Δ

Dec.	*Hexa*	*ASCII*	*Dec.*	*Hexa*	*ASCII*
128	80	Ç	154	9A	Ü
129	81	ü	155	9B	¢
130	82	é	156	9C	£
131	83	â	157	9D	¥
132	84	ä	158	9E	₧
133	85	à	159	9F	ƒ
134	86	å	160	A0	á
135	87	ç	161	A1	í
136	88	ê	162	A2	ó
137	89	ë	163	A3	ú
138	8A	è	164	A4	ñ
139	8B	ï	165	A5	Ñ
140	8C	î	166	A6	ª
141	8D	ì	167	A7	º
142	8E	Ä	168	A8	¿
143	8F	Å	169	A9	⌐
144	90	É	170	AA	¬
145	91	æ	171	AB	½
146	92	Æ	172	AC	¼
147	93	ô	173	AD	¡
148	94	ö	174	AE	«
149	95	ò	175	AF	»
150	96	û	176	B0	█
151	97	ù	177	B1	█
152	98	ÿ	178	B2	█
153	99	Ö	179	B3	│

Dec.	*Hexa*	*ASCII*	*Dec.*	*Hexa*	*ASCII*
180	B4	┤	206	CE	╬
181	B5	╡	207	CF	╧
182	B6	╢	208	D0	╨
183	B7	╖	209	D1	╤
184	B8	╕	210	D2	╥
185	B9	╣	211	D3	╙
186	BA	║	212	D4	╘
187	BB	╗	213	D5	╒
188	BC	╝	214	D6	╓
189	BD	╜	215	D7	╫
190	BE	╛	216	D8	╪
191	BF	┐	217	D9	┘
192	C0	└	218	DA	┌
193	C1	┴	219	DB	█
194	C2	┬	220	DC	▄
195	C3	├	221	DD	▌
196	C4	─	222	DE	▐
197	C5	┼	223	DF	▀
198	C6	╞	224	E0	α
199	C7	╥	225	E1	β
200	C8	╙	226	E2	Γ
201	C9	╔	227	E3	π
202	CA	╩	228	E4	Σ
203	CB	╦	229	E5	σ
204	CC	╠	230	E6	µ
205	CD	═	231	E7	γ

Dec.	*Hexa*	*ASCII*	*Dec.*	*Hexa*	*ASCII*
232	E8	ΦÝ	244	F4	⌠
233	E9	θ	245	F5	⌡
234	EA	Ω	246	F6	÷
235	EB	δ	247	F7	≈
236	EC	∞	248	F8	°
237	ED	Ø	249	F9	•
238	EE	∈	250	FA	·
239	EF	∩	251	FB	√
240	F0	≡	252	FC	^{n}
241	F1	±	253	FD	^{2}
242	F2	≥	254	FE	■
243	F3	≤	255	FF	

B

Mots réservés

Les identificateurs dont la liste suit sont des mots clés réservés du langage C. Ils ne doivent pas être utilisés pour autre chose dans un programme C. Sauf, bien entendu, à l'intérieur d'une chaîne de caractères, encadrés de guillemets. Cette liste est suivie d'une liste de mots clés du C++. Néanmoins, dans le souci d'une évolution possible de vos programmes, évitez-les.

Mot clé	*Description*
`asm`	Signale que des instructions en assembleur vont suivre.
`auto`	Classe de déclaration des variables par défaut.
`break`	Sortie forcée des constructions `for, while, switch et do...while`.
`case`	Sert à détailler les choix opérés à l'intérieur d'une instruction `switch`.
`char`	Le plus simple des types de données.
`const`	Déclare une variable qu'il est impossible de modifier.
`continue`	Permet d'ignorer la suite d'un bloc `for` ou `while` et continuer en tête de la boucle suivante.

Mot clé	*Description*
`default`	Utilisé à l'intérieur d'un `switch`. Regroupe tous les cas non explicitement spécifiés par un `case`.
`do`	Commande de boucle utilisée en conjonction avec un `while`. La boucle est toujours exécutée au moins une fois.
`double`	Type de variable pouvant contenir des valeurs exprimées en virgule flottante double précision.
`else`	Instruction précédant une alternative d'un `if`. Le bloc qui suit sera exécuté si la condition testée par le `if` n'est pas satisfaite.
`enum`	Permet de déclarer des variables n'acceptant que certaines valeurs prédéfinies.
`extern`	Modificateur indiquant qu'une variable sera déclarée dans un autre module.
`float`	Type de variable pouvant contenir des valeurs exprimées en virgule flottante simple précision.
`for`	Commande de boucle contenant les valeurs d'initialisation, incrémentation et terminaison.
`goto`	Instruction entraînant un saut vers une étiquette prédéfinie.
`if`	Modifie le déroulement du programme d'après le résultat d'une valeur `TRUE/FALSE`.
`int`	Type de variable servant à ranger des entiers.
`long`	Type de variable servant à ranger des entiers égaux ou de plus grande valeur que les `int`.
`register`	Modificateur signalant qu'une variable devrait être placée dans un registre, si c'est possible.
`return`	Dernière instruction exécutée d'une fonction. Elle permet de renvoyer un résultat à l'appelant.
`short`	Type de variable servant à contenir des entiers plus petits que les `int`.
`signed`	Modificateur signalant qu'une variable peut contenir des nombres algébriques.
`sizeof`	Opérateur retournant la taille de son argument exprimée en octets.
`static`	Modificateur signalant que la valeur d'une variable doit être conservée.
`struct`	Mot-clé servant à définir un groupement de variables de n'importe quels types.
`switch`	Tête d'aiguillage multidirectionnel utilisé en conjonction avec `case`.
`typedef`	Modificateur utilisé pour créer de nouveaux noms pour des types de variables ou de fonctions.

Mot clé	*Description*
`union`	Mot-clé permettant à plusieurs variables d'occuper le même espace de mémoire.
`unsigned`	Modificateur signalant qu'une variable peut contenir des nombres arithmétiques (positifs ou nuls).
`void`	Mot clé servant à signaler qu'une fonction ne renvoie rien ou qu'un pointeur doit être considéré comme générique, c'est-à-dire capable de pointer sur n'importe quel type de variable.
`volatile`	Modificateur signalant qu'une variable peut changer de valeur (voir `const`).
`while`	Mot clé permettant de répéter un bloc d'instructions tant que l'expression sur laquelle il porte conserve une valeur non nulle (condition vraie).

Voici une liste de mots réservés du C++ :

`catch`	`inline`	`template`
`class`	`new`	`this`
`delete`	`operator`	`throw`
`except`	`private`	`try`
`finally`	`protected`	`virtual`
`friend`	`public`	

C

Travailler avec les nombres binaires et hexadécimaux

Au cours de vos travaux de programmation, vous serez quelquefois amené à travailler avec des nombres binaires ou hexadécimaux. Ces systèmes de notation sont décrits dans cette annexe. Pour faciliter leur compréhension, nous allons commencer avec le système des nombres décimaux courant.

Le système des nombres décimaux

Le système décimal est le système de numération en base 10 le plus utilisé. Dans ce système un nombre s'exprime sous la forme de puissances de 10. Le premier chiffre (à partir de la droite) représente les puissances 0 de 10, le second chiffre donne les puissances 1 de 10, et ainsi de suite. Un nombre à la puissance 0 est toujours égal à 1 et un nombre à la puissance 1 est égal à lui-même. 342, par exemple, se décomposera de la façon suivante :

```
3   3 * 102 = 3 * 100 = 300
4   4 * 101 = 4 * 10 = 40
2   2 * 100 = 2 * 1 = 2
   Somme = 342
```

Le système de numération en base 10 utilise dix chiffres distincts de 0 à 9. Les règles qui suivent s'appliquent à tout système de numération :

- Un nombre s'exprime sous la forme de puissances de la base du système.
- Le système de numération en base *n* utilise *n* chiffres différents.

Examinons maintenant les autres systèmes de numération.

Le système binaire

Le système de numération binaire est la base 2. Il n'utilise donc que les deux chiffres 0 et 1. Ce système, très pratique pour les programmeurs, peut être utilisé pour représenter la méthode numérique sur laquelle est basé le fonctionnement de la mémoire et des puces d'ordinateur. Voici un exemple de nombre binaire avec sa représentation en notation décimale :

```
1   1 * 23 = 1 * 8 = 8
0   0 * 22 = 0 * 4 = 0
1   1 * 21 = 1 * 2 = 2
1   1 * 20 = 1 * 1 = 1
   Somme = 11 (décimal)
```

Le système binaire présente cependant un inconvénient : il n'est pas adapté à la représentation des grands nombres.

Le système hexadécimal

Le système hexadécimal est celui de la base 16. Les nombres de cette base s'expriment à l'aide de seize chiffres : les chiffres de 0 à 9 et les lettres de A à F qui représentent les

valeurs décimales 10 à 15. Voici un exemple de nombre hexadécimal avec son équivalent en notation décimale :

```
2   2 * 162 = 2 * 256 = 512
D   13 * 161 = 13 * 16 = 208
A   10 * 160 = 10 * 1 = 10
   Somme = 730 (décimal)
```

Le système hexadécimal (souvent nommé système *hexa*) est très utilisé en informatique, car il est fondé sur des puissances de 2. Chaque chiffre de ce système est équivalent à un nombre binaire de quatre chiffres et chaque nombre hexa de deux chiffres est équivalent à un nombre binaire de huit chiffres. Le Tableau C.1 présente quelques chiffres hexadécimaux et leurs équivalents en numération décimale et binaire.

Tableau C.1 : Nombres hexadécimaux avec leurs équivalents décimaux et binaires

Chiffre hexadécimal	*Équivalent décimal*	*Équivalent binaire*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	10000
F0	240	11110000



Portabilité du langage

Le terme de *portabilité* représente la possibilité pour le code source d'un programme d'être utilisé sur des plates-formes matérielles différentes. Le programme que vous avez écrit pour Linux, par exemple, pourra-t-il être compilé sur Windows, Solaris ou sur MacOS X ? Le langage C *peut* être un des langages de programmation les plus facilement portables. C'est la raison pour laquelle il est généralement préféré aux autres.

Pourquoi peut ? La plupart des compilateurs C fournissent des *extensions* avec des fonctions supplémentaires qui peuvent se révéler très utiles, mais qui ne font pas partie du standard C. Lorsque vous voudrez adapter un programme qui utilise ces extensions à une autre plate-forme, vous rencontrerez presque toujours des problèmes et vous devrez en transformer certaines parties. De même, si votre programme est un tant soit peu évolué, vous ferez peut-être appel à des bibliothèques de fonctions externes. Ces bibliothèques ont-elles été elles-mêmes portées sur les autres systèmes ? Utilisez librement les extensions de votre compilateur et les bibliothèques disponibles sur votre système si vous êtes sûr que votre programme ne sera jamais porté vers une autre plate-forme. Si la portabilité est importante, au contraire, vous devrez considérer ces extensions de façon plus approfondie et vous assurer que vous utilisez des bibliothèques assez répandues. Cette annexe souligne quelques points que vous devrez prendre en compte.

Les standards ANSI et ISO C89 et C99

La portabilité n'est obtenue que lorsque l'on adhère à un ensemble de standards suivi par les autres programmeurs et compilateurs. C'est pour cette raison qu'il est important de choisir un compilateur respectant les standards de programmation C définis par l'American National Standards Institute (ANSI) ainsi que les normes ISO C, au moins C89 (C99 n'étant pas encore nativement supportée de tous les compilateurs). Presque tous les compilateurs C offrent cette possibilité. Sachez que la norme ANSI est devenue ISO même si parler de norme ANSI est plus courant que de norme ISO. Néanmoins, la norme ISO dite ANSI a évolué pour donner une nouvelle norme ISO en 1989 appelée C89. La dernière en date est de 1999 et s'appelle C99.

Les mots clés ANSI et ISO C

Un *mot clé* est un mot réservé pour une commande de programme. Le langage C en contient relativement peu. Le Tableau D.1 vous en présente la liste.

Tableau D.1 : Les mots clés C ANSI

`asm`	`auto`	`break`
`case`	`char`	`const`
`continue`	`default`	`do`
`double`	`else`	`enum`
`extern`	`float`	`for`
`goto`	`if`	`int`
`long`	`register`	`return`
`short`	`signed`	`sizeof`
`static`	`struct`	`switch`
`typedef`	`union`	`unsigned`
`void`	`volatile`	`while`

Beaucoup de compilateurs utilisent des mots clés n'appartenant pas au tableau précédent. `near` et `huge` en sont des exemples. Ces derniers sont quelquefois reconnus par plusieurs compilateurs, mais il n'existe aucune garantie pour qu'ils soient portables vers chaque compilateur standard ANSI ou ISO C.

Reconnaissance des majuscules et des minuscules

La reconnaissance des majuscules et des minuscules est une question importante pour les langages de programmation. Contrairement certains, le langage C différencie les majuscules et les minuscules. Cela signifie qu'une variable `a` sera différente d'une variable `A`. Le Listing D.1 vous en fournit une illustration.

Listing D.1 : Reconnaissance de la casse

```
/*=======================================================*
 * Programme: listD01.c                                  *
 * Objectif: Ce programme illustre la différence entre   *
 *         les majuscules et les minuscules              *
 *=======================================================*/
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  int   var1 = 1,
        var2 = 2;
  char  VAR1 = 'A',
        VAR2 = 'B';
  float Var1 = 3.3,
        Var2 = 4.4;
  int   xyz  = 100,
        XYZ  = 500;

  printf( "\n\nImprime les valeurs des variables...\n" );
  printf( "\nLes valeurs entières:   var1 = %d, var2 = %d",
           var1, var2 );
  printf( "\nLes valeurs caractère: VAR1 = %c, VAR2 = %c",
           VAR1, VAR2 );
  printf( "\nLes valeurs flottantes:     Var1 = %f, Var2 = %f",
           Var1, Var2 );
  printf( "\nLes autres entiers:   xyz = %d, XYZ = %d",
           xyz, XYZ );

  printf( "\n\nFin de l'impression des valeurs!\n" );

  exit(EXIT_SUCCESS);
}
```

```
Imprimer les valeurs des variables...
Les valeurs entières : var1 = 1, var2 = 2
Les valeurs caractère : VAR1 = A, VAR2 = B
Les valeurs flottantes : Var1 = 3.300000, Var2 = 4.400000
Les autres entiers : xyz = 100, XYZ = 500

Fin de l'impression des valeurs !
```

Analyse

Ce programme utilise plusieurs variables de même nom. `var1` et `var2`, en lignes 10 et 11, sont définis sous la forme de valeurs entières. Ces mêmes noms de variables sont utilisés aux lignes 12 à 15 avec une casse différente. Il s'agit cette fois de valeurs caractère puis flottantes. Ces trois ensembles de déclaration placent une valeur dans les variables, ce qui permettra de les afficher aux lignes 20 à 25. Vous pouvez constater avec la sortie de ce programme que les différentes valeurs ont bien été enregistrées.

Les lignes 16 et 17 déclarent deux variables de même type et de même nom. La seule différence entre ces variables est donc l'utilisation des majuscules pour l'une et des minuscules pour l'autre. Leurs valeurs sont affichées en lignes 26 et 27.

Même si la casse peut permettre de différencier les variables, ce n'est pas une bonne pratique de programmation. Certains systèmes informatique possédant des compilateurs C ne différencient pas les majuscules des minuscules (même si cela est plutôt rare aujourd'hui). Vous risquez donc ainsi d'obtenir un code qui ne sera pas portable.

Le compilateur n'est pas le seul élément avec lequel vous pourrez rencontrer des problèmes de reconnaissance de la casse. Même si celui-ci est capable de différencier les variables, l'éditeur de lien pourrait en être incapable.

Vous avez cependant presque toujours la possibilité d'ignorer la casse des variables. La documentation de votre compilateur vous indiquera comment définir cette option. Si certaines variables ne sont différenciables que par leur casse, vous obtiendrez un message d'erreur similaire à celui-ci lorsque vous compilerez votre listing. `var1` représente bien sûr la variable utilisée :

```
listD01.c:
Error listD01.c 16: Multiple declaration for 'var1' in function main
*** 1 errors in Compile ***
```

Portabilité des caractères

L'ordinateur stocke les caractères sous la forme de nombres. Dans le cas d'un IBM PC ou d'un ordinateur compatible, la lettre A est représentée par le nombre 65 et la lettre *a* est représentée par le nombre 97. Ces nombres sont tirés d'une table ASCII (que vous trouverez en Annexe A).

Certains systèmes utilisent cependant une table différente pour les caractères de valeurs de 128 à 255. Lorsque vous créez des programmes destinés à d'autres plates-formes, vous ne pouvez donc pas être sûr que la table de traduction des caractères utilisés sera la table ASCII.



Utilisez prudemment les valeurs numériques des caractères. Ces valeurs pourraient ne pas être portables.

La définition d'un ensemble de caractères doit respecter deux règles générales. La première s'applique à la taille de la valeur d'un caractère qui ne peut excéder la taille du type `char`. Dans le cas de chaînes de caractères dont chacun est codé sur 8 bits, la valeur maximum que l'on peut enregistrer dans une variable `char` simple est 255. Vous ne devez donc pas coder un caractère avec une valeur supérieure à 255. Si vous travaillez avec des chaînes dont les caractères sont codés avec un type `wchar`, cette valeur maximum est bien plus grande.

La seconde règle à respecter est qu'un caractère soit représenté par un nombre positif. Les caractères portables appartenant à l'ensemble des caractères ASCII sont ceux qui correspondent aux valeurs 1 à 127. Il n'existe aucune garantie pour que les caractères correspondant aux valeurs 128 à 255 soient portables.

Garantir la compatibilité ANSI

La constante prédéfinie `STDC` sert, en principe, à garantir la compatibilité ANSI. Lorsqu'un listing est compilé avec l'option de compatibilité ANSI, cette constante est définie, généralement avec la valeur `1`. Dans le cas contraire, elle ne l'est pas.

Presque tous les compilateurs offrent l'option ANSI. Celle-ci est généralement disponible dans l'environnement de développement intégré ou sous la forme d'un paramètre transmis sur la ligne de commande au moment de la compilation. Cette option permet d'obtenir un programme compatible avec d'autres compilateurs et plates-formes.

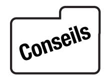


La majorité des compilateurs qui fournissent un environnement de développement intégré offrent une option ANSI. Cette option permet d'obtenir la compatibilité avec cette norme.

Le compilateur va effectuer des contrôles d'erreur supplémentaires pour assurer le respect des règles ANSI. Dans certains cas, des messages d'erreur et d'avertissement pourraient disparaître. La plupart des compilateurs, par exemple, émettent des messages d'avertissement lorsqu'une fonction est utilisée avant la définition de leur prototype. Le standard ANSI n'imposant pas l'utilisation des prototypes, ces messages d'avertissement ne seront donc plus diffusés.

Renoncer au standard ANSI

Il existe plusieurs raisons pour ne pas compiler votre programme avec l'option ANSI. Cela permettra tout d'abord de tirer parti des fonctions supplémentaires de votre compilateur. La plupart d'entre elles, comme la gestion d'écran, n'appartiennent pas au standard ANSI ou sont spécifiques au compilateur. Vous découvrirez un peu plus loin dans ce chapitre comment utiliser ces fonctions tout en conservant la compatibilité ANSI.



À faire

Différencier les variables avec d'autres critères que la casse des caractères de leurs noms.

À ne pas faire

Utiliser les valeurs numériques des caractères, à moins de savoir ce que vous faites et pourquoi vous le faites.

Les variables numériques portables

Les valeurs numériques varient en fonction des compilateurs. Le standard ANSI ne définit que quelques règles concernant ces valeurs. Le Tableau 3.2 du Chapitre 3 présentait les valeurs typiques d'un ordinateur 32 bits. Celles-ci pourraient être différentes sur certaines plates-formes.

Les types de variables suivent les règles suivantes :

- Un caractère (`char`) est le plus petit type de données. Une variable caractère (de type `char`) sera constituée d'un octet.
- Une variable courte (de type `short`) aura une taille inférieure ou égale à celle d'une variable longue (de type `long`).
- Une variable entière non signée (de type `unsigned`) aura la même taille qu'une variable entière signée (de type `int`).
- Une variable flottante (de type `float`) aura une taille inférieure ou égale à celle d'une variable double (de type `double`).

Le Listing D.2 permet d'afficher la taille des variables de la machine sur laquelle le programme est compilé.

Listing D.2 : Affichage de la taille des types de données

```
/*=========================================================*
 * Programme: listD02.c                                    *
```

```
  * Purpose: Ce programme affiche la taille des variables    *
  *    de la machine sur laquelle le programme est compilé   *
  *==========================================================*/
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  printf( "\nVariable Type Sizes" );
  printf( "\n=========================" );
  printf( "\nchar             %d", sizeof(char) );
  printf( "\nshort            %d", sizeof(short) );
  printf( "\nint              %d", sizeof(int) );
  printf( "\nlong              %d", sizeof(long) );
  printf( "\nfloat            %d", sizeof(float) );
  printf( "\ndouble           %d", sizeof(double) );

  printf( "\n\nunsigned char    %d", sizeof(unsigned char) );
  printf( "\nunsigned short   %d", sizeof(unsigned short) );
  printf( "\nunsigned int     %d\n", sizeof(unsigned int) );
  printf( "\nunsigned long     %d\n", sizeof(unsigned long) );

  exit(EXIT_SUCCESS);
}
```

On obtient un résultat du type :

```
char             1
short            2
int              4
long              4
float            4
double           8

unsigned char    1
unsigned short   2
unsigned int     4
unsigned long    4
```

Analyse

L'opérateur `sizeof()` permet d'afficher la taille de chaque type de variable en octets. Le résultat présenté correspond à une compilation du programme sur un système 32-bit avec un compilateur 32-bit. Les tailles pourront être différentes si ce même programme est compilé sur une autre plate-forme ou avec un compilateur différent. Un compilateur 64-bit, par exemple, sur une machine 64-bit pourrait donner 8 octets plutôt que 4 pour la taille d'un entier long.

Valeurs maximales et minimales

Comment déterminer les valeurs qui pourront être enregistrées si les types de variables ont des tailles différentes en fonction des machines ? Cela dépend du nombre d'octets qui constitue le type de données et si la variable est signée ou non. Le Tableau 3.2 du Chapitre 3 présente les différentes valeurs à enregistrer en fonction de ce nombre d'octets. Les valeurs maximales et minimales des types intégraux comme les entiers, sont basées sur les bits. Dans le cas des valeurs flottantes, comme les variables virgule flottante et doubles, on peut enregistrer des valeurs plus importantes si l'on renonce à la précision. Le Tableau D.2 présente les valeurs des variables intégrales et des variables en virgule flottante.

Il est intéressant de connaître la valeur maximale d'un type de variable en fonction du nombre d'octets, mais vous ignorez quelquefois ce nombre dans un programme portable. Vous ne pouvez pas non plus être sûr du niveau de précision utilisé avec les nombres en virgule flottante. Vous devez donc manipuler prudemment les nombres que vous attribuez aux variables. L'attribution d'une valeur 3000 à une variable entière, par exemple, ne pose aucun problème contrairement à l'affectation d'une valeur de 3 000 000 000. S'il s'agit d'un entier signé sur une machine 32-bit, vous obtiendrez des résultats erronés puisque la valeur maximum est de 2 147 438 647. Cette affectation ne présentera pas, en revanche, de problème dans le cas d'un entier similaire mais non signé.

Tableau D.2 : Les valeurs en fonction de la taille en octets

Nombre d'octets	*Valeurs maxi si non signé*	*Valeurs mini si signé*	*Valeurs maxi si signé*
		Types entiers	
1	255	– 128	127
2	65 535	– 32 768	32 767
4	4 294 967 295	– 2 147 483 648	2 147 438 647
8		1,844674 * E19	
		Nombres flottants	
4*		3,4 E–38	3,4 E38
8**		1,7 E–308	1,7 E308
10***		3,4 E–4932	1,1 E4932

* Précision sur 7 chiffres
** Précision sur 15 chiffres
*** Précision sur 19 chiffres



Les valeurs du Tableau D.2 pourraient être différentes sur certains compilateurs. Les nombres à virgule flottante, en particulier, peuvent différer avec leurs divers niveaux de précision. Les tableaux D.3 et D.4 permettent de fixer certaines valeurs indépendamment de la machine utilisée.

Le standard ANSI fournit un ensemble de constantes qui doivent être inclues dans les fichiers en-tête limits.h et float.h. Ces constantes définissent le nombre de bits dans un type de variable donné ainsi que les valeurs minimales et maximales de ces variables. Le Tableau D.3 fournit la liste des valeurs de limits.h. Ces dernières s'appliquent aux types de données intégraux. Les valeurs contenues dans float.h correspondent aux types virgule flottante.

Tableau D.3 : Les constantes définies par l'ANSI dans limits.h

Constante	***Valeur***
`CHAR BIT`	Le nombre de bits de la variable caractère
`CHAR MIN`	La valeur minimale (signée) de la variable caractère
`CHAR MAX`	La valeur maximale (signée) de la variable caractère
`SCHAR MIN`	La valeur minimale de la variable caractère signée
`SCHAR MAX`	La valeur maximale de la variable caractère signée
`UCHAR MAX`	La valeur maximale du caractère non signé
`INT MIN`	La valeur minimale de la variable entière
`INT MAX`	La valeur maximale de la variable entière
`UINT MAX`	La valeur maximale de la variable entière non signée
`SHRT MIN`	La valeur minimale de la variable short
`SHRT MAX`	La valeur maximale de la variable short
`USHRT MAX`	La valeur maximale de la variable short non signée
`LONG MIN`	La valeur minimale de la variable long
`LONG MAX`	La valeur maximale de la variable de type long
`ULONG MAX`	La valeur maximale de la variable de type long non signée

Les valeurs des Tableaux D.3 et D.4 peuvent être utilisées pour enregistrer des nombres. Vous obtiendrez un code portable en vérifiant que le nombre est supérieur ou égal à la constante minimale et inférieur ou égal à la constante maximale. Le listing D.3 affiche les

Tableau D.4 : Les constantes définies par l'ANSI dans float.h

Constante	*Valeur*
`FLT DIG`	Les chiffres de précision d'une variable de type `float`
`DBL DIG`	Les chiffres de précision d'une variable de type `double`
`LDBL DIG`	Les chiffres de précision d'une variable de type `long double`
`FLT MAX`	La valeur maximale de la variable virgule flottante
`FLT MAX 10 EXP`	La valeur maximale de l'exposant de la variable virgule flottante (base 10)
`FLT MAX EXP`	La valeur maximale de l'exposant de la variable virgule flottante (base 2)
`FLT MIN`	La valeur minimale de la variable virgule flottante
`FLT MIN 10 EXP`	La valeur minimale de l'exposant de la variable virgule flottante (base 10)
`FLT MIN EXP`	La valeur minimale de l'exposant de la variable virgule flottante (base 2)
`DBL MAX`	La valeur maximale d'une variable de type double
`DBL MAX 10 EXP`	La valeur maximale de l'exposant de la variable de type double (base 10)
`DBL MAX EXP`	La valeur maximale de l'exposant de la variable de type double (base 2)
`DBL MIN`	La valeur minimale d'une variable double
`DBL MIN 10 EXP`	La valeur minimale de l'exposant d'une variable double (base 10)
`DBL MIN EXP`	La valeur minimale de l'exposant d'une variable double (base 2)
`LDBL MAX`	La valeur maximale d'une variable de type long double
`LDBL MAX 10 DBL`	La valeur maximale de l'exposant d'une variable long double (base 10)
`LDBL MAX EXP`	La valeur maximale de l'exposant d'une variable long double (base 2)
`LDBL MIN`	La valeur minimale d'une variable long double
`LDBL MIN 10 EXP`	La valeur minimale de l'exposant d'une variable long double (base 10)
`LDBL MIN EXP`	La valeur minimale de l'exposant d'une variable long double (base 2)

valeurs des constantes définies ANSI et le Listing D.4 met en œuvre quelques-unes de ces constantes. Il est possible que vous obteniez un résultat différent selon le compilateur utilisé.

Listing D.3 : Affichage des valeurs des constantes ANSI

```
/*=====================================================*
 * Programme:  listD03.c                               *
 * Affichage des constantes définies.                  *
 *=====================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <limits.h>

int main( void )
{
    printf( "\n CHAR_BIT        %d ", CHAR_BIT );
    printf( "\n CHAR_MIN        %d ", CHAR_MIN );
    printf( "\n CHAR_MAX        %d ", CHAR_MAX );
    printf( "\n SCHAR_MIN       %d ", SCHAR_MIN );
    printf( "\n SCHAR_MAX       %d ", SCHAR_MAX );
    printf( "\n UCHAR_MAX       %d ", UCHAR_MAX );
    printf( "\n SHRT_MIN        %d ", SHRT_MIN );
    printf( "\n SHRT_MAX        %d ", SHRT_MAX );
    printf( "\n USHRT_MAX       %d ", USHRT_MAX );
    printf( "\n INT_MIN         %d ", INT_MIN );
    printf( "\n INT_MAX         %d ", INT_MAX );
    printf( "\n UINT_MAX        %e ", (double)UINT_MAX );
    printf( "\n LONG_MIN        %ld ", LONG_MIN );
    printf( "\n LONG_MAX        %ld ", LONG_MAX );
    printf( "\n ULONG_MAX       %e ", (double)ULONG_MAX );
    printf( "\n FLT_DIG         %d ", FLT_DIG );
    printf( "\n DBL_DIG         %d ", DBL_DIG );
    printf( "\n LDBL_DIG        %d ", LDBL_DIG );
    printf( "\n FLT_MAX         %e ", FLT_MAX );
    printf( "\n FLT_MIN         %e ", FLT_MIN );
    printf( "\n DBL_MAX         %e ", DBL_MAX );
    printf( "\n DBL_MIN         %e \n", DBL_MIN );

    exit(EXIT_SUCCESS);
}
```

Vous obtenez le résultat suivant :

```
CHAR_BIT        8
CHAR_MIN        -128
CHAR_MAX        127
SCHAR_MIN       -128
SCHAR_MAX       127
UCHAR_MAX       255
SHRT_MIN        -32768
SHRT_MAX        32767
USHRT_MAX       65535
INT_MIN         -2147483648
INT_MAX         2147483647
UINT_MAX        4.294967e+09
LONG_MIN        -2147483648
```

```
LONG_MAX        2147483647
ULONG_MAX       4.294967e+09
FLT_DIG         6
DBL_DIG         15
LDBL_DIG        18
FLT_MAX         3.402823e+38
FLT_MIN         1.175494e-38
DBL_MAX         1.797693e+308
DBL_MIN         2.225074e-308
```



Les valeurs obtenues pourront varier en fonction du compilateur. Avec les machines 64 bits et la compatibilité 32 bits de celles-ci, il est parfois possible de paramétrer le compilateur sur la taille des types et leurs limites.

Analyse

Ce programme très simple réalise quelques appels de fonction `printf()`. Chacun de ces appels affiche une constante définie différente. Vous remarquerez que le caractère de conversion est adapté au type de valeur à afficher. Vous obtenez ainsi un résumé des valeurs utilisées par votre compilateur. Vous auriez pu aussi trouver ces valeurs en consultant les fichiers en-tête float.h et limits.h.

Listing D.4 : Utilisation des constantes ANSI

```
/*===============================================================*
 * Programme: listD04.c                                          *
 *                                                               *
 * Objectif: Utiliser les constantes maximales et minimales.
 *                                                               *
 * Note:    On ne peut pas afficher tous les caractères valides *
 *          à l'écran!                                           *
 *===============================================================*/

#include <float.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    unsigned char ch;
    int  i;

    printf( "Entrez une valeur numérique.");
    printf( "\nCette valeur sera convertie en caractère.");
    printf( "\n\n==> " );

    scanf("%d", &i);
```

```
25:
26:      while( i < 0 || i > UCHAR_MAX )
27:      {
28:         printf("\n\nValeur incorrecte pour un caractère.");
29:         printf("\nEntrez une valeur entre 0 et %d ==> ", UCHAR_MAX);
30:
31:         scanf("%d", &i);
32:      }
33:      ch = (char) i;
34:
35:      printf("\n\n%d représente le caractère %c\n", ch, ch );
36:
37:      exit(EXIT_SUCCESS);
38:  }
```

Vous obtenez l'affichage suivant :

```
Entrez une valeur numérique.
Cette valeur sera convertie en caractère.

==> 5000


Valeur incorrecte pour un caractère.
Entrez une valeur entre 0 et 255 ==> 69


69 représente le caractère E
```

Analyse

Le Listing D.4 illustre l'utilisation de la constante `UCHAR MAX`. Les premiers éléments à remarquer sont les deux fichiers include contenant les constantes définies des lignes 10 et 11. Le fichier float.h n'est pas nécessaire puisque vous n'utilisez aucune constante virgule flottante. La ligne 11 est cependant indispensable, car le fichier en-tête contient la définition de `UCHAR MAX`.

Les variables du programme sont déclarées en lignes 17 et 18. Plusieurs instructions `scanf` interrogent ensuite l'utilisateur pour obtenir un nombre. La variable qui recevra ce dernier est un entier, car il pourra enregistrer un nombre important. Si vous aviez choisi un type caractère, un nombre trop grand aurait du être tronqué pour s'y adapter. Vous pouvez facilement le vérifier en transformant le `i` de la ligne 24 en `ch`.

La ligne 26 fait appel à la constante définie pour tester le nombre saisi et le comparer à la valeur maximale d'un caractère non signé. Si la valeur saisie par l'utilisateur n'est pas valide pour ce type de variable (caractère non signé), un message indiquera les valeurs correctes et imposera la saisie de l'une d'entre elles.

La ligne 33 converti l'entier en caractère. Dans un programme plus complexe, cette conversion simplifierait considérablement les opérations suivantes. Vous éviteriez ainsi de réallouer une valeur incorrecte à un caractère dans la variable entière. Dans le cadre de ce programme, la ligne 35 qui affiche le caractère obtenu aurait pu utiliser `i` à la place de `ch`.

Classification des nombres

Il est quelquefois nécessaire d'obtenir des informations concernant une variable. Vous pourriez avoir besoin de savoir, par exemple, si l'information est numérique, un caractère de contrôle, un caractère majuscule, ou une autre classification parmi la douzaine existante. Il existe deux méthodes pour effectuer ce type de contrôle. Le Listing D.5 vous présente une méthode permettant de déterminer si la valeur enregistrée dans un type caractère est une lettre de l'alphabet.

Listing D.5 : Le type caractère est-il une lettre de l'alphabet ?

```
/*============================================================*
 * Programme: listD05.c                                       *
 * Objectif: La façon dont ce programme utilise les valeurs  *
 *          caractère n'est pas portable.                     *
 *============================================================*/
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   unsigned char x = 0;
   char trash[256];          /* Supprime les touches supplémentaires */
   while( x != 'Q' && x != 'q' )
   {
      printf( "\n\nEntrez un caractère (Q pour quitter) ==> " );

      x = getchar();

      if( x >= 'A' && x <= 'Z')
      {
         printf( "\n\n%c est une lettre de l'alphabet!", x );
         printf("\n%c est une lettre majuscule!", x );
      }
      else
      {
         if( x >= 'a' && x <= 'z')
         {
            printf( "\n\n%c est une lettre de l'alphabet!", x );
            printf("\n%c est une lettre minuscule!", x );
         }
         else
```

```
            {
               printf( "\n\n%c n'est pas une lettre de l'alphabet!", x );
            }
         }
         lire_clavier(trash, sizeof(trash)); /* Supprime la touche entrée */
      }
      printf("\n\nMerci d'avoir participé!\n");
      exit(EXIT_SUCCESS);
   }
```

Vous obtenez le résultat suivant :

```
Entrez un caractère (Q pour quitter) ==> A

A est une lettre de l'alphabet!
A est une lettre majuscule!

Entrez un caractère (Q pour quitter) ==> f

f est une lettre de l'alphabet!
f est une lettre minuscule!

Entrez un caractère (Q pour quitter) ==> 1

1 n'est pas une lettre de l'alphabet!

Entrez un caractère (Q pour quitter) ==> *

* n'est pas une lettre de l'alphabet!

Entrez un caractère (Q pour quitter) ==> q

q est une lettre de l'alphabet!
q est une lettre minuscule!

Merci d'avoir participé!
```

Analyse

Ce programme contrôle si la lettre est située entre le A et le Z majuscule ou entre le *a* et le *z* minuscule. Si la lettre se situe entre ces deux intervalles, vous pouvez conclure qu'il s'agit d'une lettre alphabétique. Cependant, cette comparaison est un peu lourde à mettre en œuvre et le devient encore plus si vous souhaitez tester s'il s'agit d'un blanc, d'un chiffre... Il est conseillé d'utiliser une fonction de test de type de caractère.

Il existe plusieurs fonctions de ce type. Le Tableau D.5 vous en présente la liste avec le type de contrôle effectué. Ces fonctions renvoient la valeur 0 si le caractère concerné ne satisfait pas au contrôle. Elle renvoie une valeur non nulle dans le cas contraire.

Tableau D.5 : Les fonctions de classification de caractère

Fonction	*Description*
`isalnum()`	Vérifie si le caractère est alphanumérique
`isalpha()`	Vérifie si le caractère est alphabétique
`iscntrl()`	Vérifie si le caractère est un caractère de contrôle
`isdigit()`	Vérifie si le caractère est un chiffre décimal
`isgraph()`	Vérifie si le caractère est imprimable (l'espace est une exception)
`islower()`	Vérifie si le caractère est une minuscule
`isprint()`	Vérifie si le caractère est imprimable
`ispunct()`	Vérifie si le caractère est un caractère de ponctuation
`isspace()`	Vérifie si le caractère est un caractère blanc
`isupper()`	Vérifie si le caractère est une majuscule
`isxdigit()`	Vérifie si le caractère est un chiffre hexadécimal

Il est déconseillé de comparer les valeurs de deux caractères différents, sauf pour en contrôler l'égalité. Vous pouvez contrôler, par exemple, si la valeur d'une variable caractère est égale à `'A'`, mais vous ne devez pas vérifier si la valeur d'un caractère est supérieure à `'A'`, même si d'autres le font (c'est malheureusement une pratique courante).

```
if( X > 'A' )     /* DECONSEILLE */
...
if( X == 'A' )    /* CORRECT */
...
```

Le programme du Listing D.6 reprend celui du Listing D.5 en utilisant les valeurs de classification de caractère appropriées.

Listing D.6 : Utilisation des fonctions de classification de caractère

```
/*============================================================*
 * Programme: listD06.c                                       *
 *                                                            *
```

```
 * Objectif: Ce programme constitue une alternative à celui     *
 *                    du Listing D.5.                           *
 *==============================================================*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(void)
{
  unsigned char x = 0;
  char trash[256];          /* supprime les touches supplémentaires */
  while( x != 'Q' && x != 'q' )
  {
     printf( "\n\nEntrez un caractère (Q pour quitter) ==> " );

     x = getchar();

     if( isalpha(x) )
     {
        printf( "\n\n%c est une lettre de l'alphabet!", x );
        if( isupper(x) )
        {
           printf("\n%c est une lettre majuscule!", x );
        }
        else
        {
           printf("\n%c est une lettre minuscule!", x );
        }
     }
     else
     {
        printf( "\n\n%c n'est pas une lettre de l'alphabet!", x );
     }
     lire_clavier(trash, sizeof(trash));   /* Insère les touches supplémentaires */
  }
  printf("\n\nMerci d'avoir participé!\n");
  exit(EXIT_SUCCESS);
}
```

L'exécution de ce programme donne le résultat suivant :

```
Entrez un caractère (Q pour quitter) ==> z

z est une lettre de l'alphabet!
z est une lettre minuscule!

Entrez un caractère (Q pour quitter) ==> T

T est une lettre de l'alphabet!
T est une lettre majuscule!

Entrez un caractère (Q pour quitter) ==> #

# n'est pas une lettre de l'alphabet!

Entrez un caractère (Q pour quitter) ==> 7
```

```
7 n'est pas une lettre de l'alphabet!

Enter un caractère (Q pour quitter) ==> Q

Q est une lettre de l'alphabet!
Q est une lettre majuscule!

Merci d'avoir participé!
```

Analyse

Si vous avez utilisé les mêmes valeurs, le résultat obtenu devrait être identique à celui du Listing D.5. Ce programme a mis en œuvre les fonctions de classification de caractère. Remarquez le fichier en-tête ctype.h de la ligne 8 qui introduit ces fonctions. La ligne 20 fait appel à la fonction `isalpha()` pour contrôler si le caractère saisi est une lettre de l'alphabet. Elle affiche dans ce cas le message de la ligne 22. La ligne 23 contrôle ensuite si le caractère est une majuscule à l'aide de la fonction `isupper()`. Le message affiché est celui de la ligne 25 si le caractère est effectivement une majuscule et celui de la ligne 29 dans le cas contraire. Si le caractère n'est pas une lettre de l'alphabet, le message de la ligne 34 est envoyé. L'exécution du programme se poursuivra jusqu'à l'activation de la touche Q ou q grâce à la boucle `while` qui débute en ligne 14.



À ne pas faire

Utiliser les valeurs numériques lorsque vous recherchez les variables maximales. Utilisez les constantes définies si vous voulez créer un programme portable.

À faire

Utiliser les fonctions de classification de caractère chaque fois que c'est possible.

Souvenez-vous que (`!=`) est considéré comme un contrôle d'égalité.

Exemple de portabilité : la conversion de la casse d'un caractère

La conversion de la casse d'un caractère est une opération courante en programmation. On fait généralement appel à une fonction du type :

```
char conv_maj( char x )
{
if( x >= 'a' && x <= 'z' )
{
x -= 32;
}
return( x )
}
```

Cette manière de faire pose un problème car non seulement on devrait tester avec `isalpha()` si le caractère est alphanumérique et avec `islower()` s'il s'agit de minuscules, mais aussi et surtout, le cas des caractères accentués n'est pas traité.

Deux fonctions standards ANSI permettent de modifier la casse d'un caractère : `toupper()` convertit un caractère minuscule en majuscule et `lowercase()` convertit un caractère majuscule en minuscule. La ligne qui suit peut remplacer la fonction précédente :

```
toupper();
```

Cette fonction n'a pas besoin d'être créée puisqu'elle existe déjà. Cette fonction étant définie par le standard ANSI, elle devrait être portable. Il n'est pas dit qu'elle convertisse les caractères accentués mais, si elle le fait, elle le fera bien.

Unions et structures portables

La portabilité doit aussi être étudiée lorsque lors de la création de structures et d'unions. L'alignement des mots et l'ordre dans lequel les membres sont stockés peuvent aussi poser des problèmes d'incompatibilité.

Alignement des mots

L'alignement des mots est un facteur important de la portabilité d'une structure. Un *mot* est un nombre d'octets défini généralement équivalent à la taille du processeur de l'ordinateur utilisé. Un ordinateur 32-bit, par exemple, utilise des mots de 4 octets (43 bits).

Ces explications seront plus claires avec un exemple. Examinez la structure suivante. Le nombre d'octets nécessaire pour enregistrer cette structure est déterminé en calculant le nombre d'entiers de 4 octets et le nombre de caractères sur un octet.

```
struct struct_tag {
int    x;    /* Les int occupent 4 octets */
char   a;    /* Les char occupent 1 octet */
int    y;
char   b;
int    z;
} sample = { 100, 'A', 200, 'B', 300);
```

Après un premier calcul, 18 octets de mémoire seront nécessaires, mais la mémoire réelle utilisée pourra être supérieure. Si l'alignement des mots est activé, cette structure occupera 20 octets. Les Figures D.1 et D.2 illustrent les possibilités d'enregistrement de cette structure dans la mémoire.

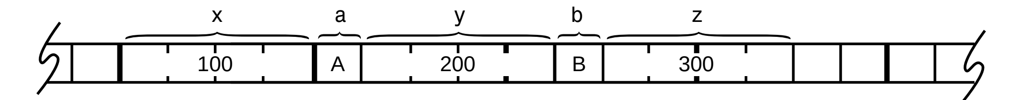


Figure D.1
L'alignement des mots est désactivé.

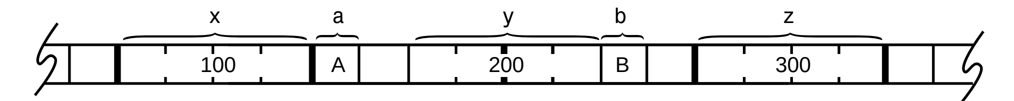


Figure D.2
L'alignement des mots est activé.

Un programme ne peut savoir si l'alignement des mots sera ou non réalisé. Les membres pourront être alignés sur chaque groupe de 2 octets, de 4 octets ou de 8 octets. Il n'y a aucun moyen de prévoir la technique utilisée.

Lire et créer des structures

Vous devez respecter quelques règles lors de la lecture ou de la création de structures. Essayez de ne plus utiliser de constante littérale pour stocker la taille d'une structure ou d'une union. Le fichier à partir duquel vous lisez ou créez des structures risque de ne pas être portable. Il convient donc de rendre le programme portable et celui-ci lira et écrira ensuite les fichiers de données spécifiques à la machine sur laquelle il a été compilé. L'exemple qui suit présente une instruction `read` qui devrait être portable :

```
fread( &the_struct, sizeof( the_struct ), 1, filepointer );
```

La constante littérale a été remplacée par la commande `sizeof`. Quel que soit l'alignement des octets en cours, on les lira ainsi.

Les directives du préprocesseur

Vous avez étudié plusieurs directives du préprocesseur au Chapitre 21. Plusieurs d'entre elles ont été définies dans le standard ANSI. Le Tableau D.6 vous présente ces dernières.

Tableau D.6 : Les directives de préprocesseur du standard ANSI

`#define`	`#if`
`#elif`	`#ifdef`
`#else`	`#ifndef`
`#endif`	`#include`
`#error`	`#pragma`

Utiliser des constantes prédéfinies

La plus grande partie des constantes prédéfinies sont spécifiques au compilateur qui vous les propose. Elles ne sont donc probablement pas portables. Le standard ANSI en définit plusieurs dont les suivantes :

Constantes	*Description*
`DATE`	Cette constante sera remplacée par la date à laquelle le programme a été compilé. Cette date est une chaîne littérale au format `"Mmm JJ, AAA"`. Le 16 janvier 2008 apparaîtra donc sous la forme `"Jan 16, 2008"`.
`FILE`	Une chaîne littérale correspondant au nom du fichier source au moment de la compilation.
`LINE`	Une valeur décimale représentant le numéro de la ligne du code source sur laquelle apparaît `LINE` .
`STDC`	Cette constante sera définie avec la valeur `1` si le fichier source est compilé en suivant le standard ANSI. Dans le cas contraire, elle ne sera pas définie.
`TIME`	Une chaîne constante représentant l'heure à laquelle le programme a été compilé. Le format est `"HH:MM:SS"`.

Utiliser des éléments non ANSI dans des programmes portables

Un programme peut utiliser des constantes et toute autre commande non définie dans le standard ANSI tout en restant portable. Il suffit pour cela de limiter l'utilisation de ces constantes aux compilateurs supportant les fonctions utilisées. La plupart des compilateurs fournissent des constantes définies qui permettent de les identifier. En définissant des zones de code supportées par chacun des ces compilateurs, vous pouvez obtenir un programme portable. Le Listing D.7 illustre une programmation de ce type.

Listing D.7 : Un programme portable avec des zones spécifiques aux compilateurs

```
 /*========================================================*
  * Programme: listD07.c                                    *
  * Objectif: Ce programme illustre l'utilisation des       *
  *                   constantes définies.                  *
  * Note:    Ce programme donne des résultats différents    *
  *          en fonction du compilateur utilisé.            *
  *========================================================*/
#include <stdio.h>
#include <stdlib.h>

#ifdef MS_WINDOWS
```

Listing D.7 : Un programme portable avec des zones spécifiques aux compilateurs *(suite)*

```

#define STRING "CREATION D'UN PROGRAMME WINDOWS!\n"

#else

#define STRING "LE PROGRAMME EN COURS N'EST PAS UN PROGRAMME WINDOWS \n"

#endif

int main(void)
  {
    printf( "\n\n") ;
    printf( STRING );

#ifdef _MSC_VER

    printf( "\n\nUtilisation d'un compilateur de Microsoft!" );
    printf( "\n   La version de votre compilateur est %s\n", _MSC_VER );

#endif

#ifdef __TURBOC__

   printf( "\n\nUtilisation du compilateur Turbo C!" );
   printf( "\n   Votre version de compilateur est %x\n", __TURBOC__ );

#endif

#ifdef __BORLANDC__

   printf( "\n\nUtilisation d'un compilateur Borland!\n" );

#endif

#ifdef __GNUC__

   printf( "\n\nUtilisation du compilateur GCC!\n" );

#endif

   exit(EXIT_SUCCESS);
  }
```

Voici le résultat obtenu en exécutant ce programme après une compilation avec Turbo C 3.0 pour DOS :

```
LE PROGRAMME EN COURS N'EST PAS UN PROGRAMME WINDOWS
```

```
Utilisation du compilateur Turbo C!
   Votre version de compilateur est 300
```

Le résultat qui suit est obtenu en exécutant le programme avec le compilateur Borland C++ sous DOS :

```
LE PROGRAMME EN COURS N'EST PAS UN PROGRAMME WINDOWS

Utilisation d'un compilateur Borland!
```

Le résultat qui suit est obtenu en exécutant le programme avec un compilateur Microsoft sous DOS :

```
LE PROGRAMME EN COURS N'EST PAS UN PROGRAMME WINDOWS

Utilisation d'un compilateur de Microsoft!
   La version de votre compilateur est >>
```

Le résultat qui suit est obtenu en exécutant le programme avec le compilateur GCC sous Linux :

```
LE PROGRAMME EN COURS N'EST PAS UN PROGRAMME WINDOWS

Utilisation du compilateur GCC!
```

Analyse

Ce programme utilise les constantes définies pour reconnaître le compilateur utilisé. La directive du préprocesseur `#ifdef` de la ligne 11 vérifie si la constante qui suit est bien définie. Dans ce cas, les instructions suivantes seront exécutées jusqu'à l'apparition de la directive `#endif`. La constante `STRING` reçoit alors le message approprié qui est ensuite affiché par la ligne 24.

La ligne 26 vérifie la définition de `MSC VER`. Cette constante contient le numéro de version d'un compilateur Microsoft. Si le compilateur est d'un autre type, elle ne sera pas définie. La ligne 29 affichera l'information concernant cette version après que la ligne 28 ait annoncé l'utilisation d'un compilateur Microsoft.

Les lignes 33 à 38, 40 à 44 et 46 à 50 se comportent de façon analogue. Elles contrôlent l'utilisation d'un compilateur professionnel, du Turbo C de Borland ou du compilateur GCC.

Ce programme s'appuie sur les constantes pour reconnaître le compilateur. Le message obtenu est le même quel que soit le compilateur utilisé. Si vous connaissez les systèmes vers lesquels votre programme devra être porté, vous pourrez introduire dans votre code les commandes qui leur sont spécifiques. Dans ce cas, vous devrez fournir le code approprié pour chaque compilateur.

Petit boutiste et gros boutiste (Little Endian et Big Endian)

Sous ces termes se cache la représentation des mots (blocs de 2, 4 ou 8 octets) en mémoire RAM. Cette représentation n'est pas la même sur toutes les machines. Par exemple, pour une machine 32 bits (les mots sont donc de 4 octets), il existe deux façons courantes de stocker un mot. Prenons le nombre hexadécimal "11223344" (il ne s'agit pas de chaînes de caractères ici). Il peut être codé soit "11223344" comme cela paraît évident, mais aussi "33441122". Si cela vous surprend, vous allez être encore plus étonné de savoir que c'est de cette façon que procèdent les PC (architecture i386).

Lorsque vous mettez "11223344" en mémoire, les ordinateurs dits petits boutistes le codent "11223344" et le restituent "11223344". Sur les ordinateurs dits gros boutistes, "11223344" est codé "33441122" et décodé "11223344", ce qui rend transparente la chose si vous travaillez sur un seul ordinateur ou sur des ordinateurs de même type.

Les choses se corsent lorsque des données sont échangées (par des fichiers par exemple) entre des ordinateurs de type différent. Ainsi, pour transférer notre "11223344", un ordinateur gros boutiste le codera "33441122" comme nous venons de le voir. Après le transfert sur un ordinateur petit boutiste qui aura reçu "33441122", notre donnée sera décodée telle quelle, soit "33441122" au lieu du "11223344" attendu.

Pour garantir la portabilité des données de vos programmes, vous devez tenir compte du boutiste de votre ordinateur et de ceux des autres. Il y a deux façons de s'en sortir. L'une consiste à définir un boutiste de transfert. Par exemple, vous pouvez décider d'utiliser le petit boutiste. Sur un ordinateur petit boutiste, la conversion reviendra à ne rien faire. Sur un ordinateur gros boutiste, il faudra inverser chaque moitié du mot. La seconde façon de faire est de tout simplement éviter le problème en ne stockant que des chaînes de caractères. En effet, une chaîne n'est rien d'autre qu'un tableau de caractères, soit des espaces de un octet. Les caractères ne sont donc pas impactés par le boutiste de l'ordinateur. Par exemple, plutôt que d'écrire notre nombre "11223344" sous forme d'un entier long, il suffirait de le convertir en chaîne de caractères (avec `sprintf()` ou `fprintf()` selon le cas) avant de le transférer. Après transfert, la chaîne de caractère lue serait transformée en entier long (avec `strtol()`).

Pour éviter les problèmes de boutiste, il est conseillé de transférer les données au format texte, en convertissant les données si nécessaire. Cela présente également un autre avantage majeur : lorsque le transfert s'effectue *via* un fichier, celui-ci peut être lu avec un simple éditeur de texte. Cela facilite d'une part le débogage et d'autre part l'interopérabilité entre programmes en favorisant l'écriture d'autres programmes qui auraient un meilleur accès au format du fichier.

Les fichiers en-tête du standard ANSI

Le standard ANSI définit plusieurs fichiers d'en-tête que vous pouvez inclure pour créer des programmes portables. Ces fichiers sont décrits en Annexe E.

Résumé

Cette annexe est riche en informations concernant la portabilité. Le langage C est un des langages les plus facilement portables, mais cela implique le respect de quelques règles. Le standard ANSI a été créé pour qu'un programme C puisse être utilisé quel que soit le compilateur ou le système. Les règles à respecter pour obtenir un code portable concernent la casse des variables, le choix de l'ensemble des caractères à utiliser, l'utilisation de valeurs numériques portables, la vérification des tailles de variables, la comparaison des caractères, l'utilisation des structures et des unions, et l'utilisation des directives et des constantes du préprocesseur. Pour terminer, vous avez étudié comment introduire du code spécifique à un compilateur dans un programme portable.

Q & R

Q Comment peut-on créer des programmes graphique portables ?

R La programmation des graphiques n'est pas prise en compte par le standard ANSI. Vous devez utiliser des bibliothèques graphiques qui ont été portées sur diverses architectures. Citons entre autres la bibliothèque GTK+ qui existe sur de nombreux Unix, Linux et Windows.

Q La portabilité doit-elle toujours être recherchée ?

R Il n'est pas toujours nécessaire de se soucier de portabilité. Certains programmes peuvent être uniquement destinés à votre propre utilisation et n'auront donc pas besoin d'être portés vers un système différent. Vous pourrez utiliser dans ce cas des fonctions non portables comme `system()`.

Q Les commentaires sont-ils portables s'ils sont précédés de // plutôt que situés entre /* et */ ?

R Non, les commentaires précédés de deux slashs sont des commentaires C++. Ils sont maintenant utilisés par de nombreux programmeurs C, mais ils ne sont pas encore définis dans les standards. Il est donc possible que certains compilateurs ne les supportent pas.

Atelier

Les questions et exercices qui suivent ne sont pas corrigés en annexe.

Questionnaire

1. Quel est le critère le plus important entre l'efficacité et la facilité de maintenance ?
2. Quelle est la valeur numérique de la lettre a ?
3. Quelle est la plus grande valeur caractère non signée sur votre système ?
4. Que signifie ANSI ?
5. Les noms de variable suivants sont-ils valides dans le même programme C ?

```
int lastname,
LASTNAME,
LastName,
Lastname;
```

6. Que fait `isalpha()` ?
7. Que fait `isdigit()` ?
8. Quel intérêt représente l'utilisation de fonctions telles que `isalpha()` ou `isdigit()` ?
9. L'enregistrement des structures sur le disque est-il indépendant de la portabilité ?
10. La constante `TIME` peut-elle être utilisée dans une instruction `printf()` pour afficher l'heure courante ? Voici un exemple :

```
printf( "L'heure courante est:  %s", __TIME__ );
```

Exercices

1. **CHERCHEZ L'ERREUR :** La fonction suivante est-elle correcte ?

```
void Print_error( char *msg )
{
    static int ctr = 0,
               CTR = 0;
    printf("\n" );
    for( ctr = 0; ctr < 60; ctr++ )
    {
       printf("*");
    }
```

```
    printf( "\nError %d, %s - %d: %s.\n", CTR,
           __FILE__, __LINE__, msg );
    for( ctr = 0; ctr < 60; ctr++ )
    {
        printf("*");
    }
}
```

2. Créez une fonction qui vérifie si un caractère est une voyelle.
3. Écrivez une fonction renvoyant `0` si elle reçoit un caractère qui n'est pas une lettre de l'alphabet, `1` s'il s'agit d'une lettre majuscule, et `2` s'il s'agit d'une minuscule. Créez cette fonction aussi standard que possible.
4. Recherchez les indicateurs de votre compilateur à définir pour ignorer la casse des variables, activer l'alignement des octets et garantir la compatibilité ANSI.
5. Le code suivant est-il portable ?

```
void list_a_file( char *nom_fichier )
{
    system("TYPE" file_name );
}
```

6. Le code suivant est-il portable ?

```
int to_upper( int x )
{
    if( x >= 'a' && x <= 'z' )
    {
        toupper( x );
    }
    return( x );
}
```



Fonctions C courantes

Dans cette annexe, vous trouverez la liste des prototypes des fonctions contenues dans chacun des fichiers d'en-tête fournis avec la plupart des compilateurs C. Celles à côté desquelles figure un astérisque ont été traitées dans ce livre.

Les fonctions sont listées par ordre alphabétique. A la suite du nom, vous trouverez le prototype complet. Vous remarquerez que la notation employée diffère de celle que nous avons utilisée tout au long de cet ouvrage. Seul figure le type des arguments utilisés ; aucun nom ne leur est donné. En voici deux exemples :

```
int func1(int, int*);
int func1(int x, int *y);
```

Dans les deux cas, on déclare une fonction `func1` qui accepte deux arguments : le premier est de type `int`, le second est un pointeur vers un type `int`. Pour le compilateur, ces deux déclarations sont équivalentes.

Tableau E.1 : Fonctions C courantes listées par ordre alphabétique

Fonction	*Fichier d'en-tête*	*Prototype de la fonction*
`abort*`	stdlib.h	`void abort(void);`
`abs`	stdlib.h	`int abs(int);`
`acos*`	math.h	`double acos(double);`
`asctime*`	time.h	`char *asctime(const struct tm *);`
`asin*`	math.h	`double asin(double);`
`assert*`	assert.h	`void assert(int);`
`atan*`	math.h	`double atan(double);`
`atan2*`	math.h	`double atan2 (double, double);`
`atexit*`	stdlib.h	`int atexit(void (*)(void));`
`atof*`	stdlib.h	`double atof(const char *);`
`atof*`	math.h	`double atof(const char *);`
`atoi*`	stdlib.h	`int atoi(const char *);`
`atol*`	stdlib.h	`long atol(const char *);`
`bsearch*`	stdlib.h	`void *bsearch(const void *, size t, size t,` `(const void*, const void*));`
`calloc*`	stdlib.h	`void *calloc(size t, size t);`
`ceil*`	math.h	`double ceil (double);`
`clearerr`	stdio.h	`void clearerr(FILE *);`
`clock*`	time.h	`clock t clock(void);`
`cos*`	math.h	`double cos(double);`
`cosh*`	math.h	`double cosh(double);`
`ctime*`	time.h	`char *ctime(const time t *);`
`difftime`	time.h	`double diff time(time t, time t);`
`div`	stdlib.h	`div t div(int, int);`
`exit*`	stdlib.h	`void exit(int);`
`exp*`	math.h	`double exp(double);`

Tableau E.1 : Fonctions C courantes listées par ordre alphabétique *(suite)*

Fonction	*Fichier d'en-tête*	*Prototype de la fonction*
`fabs*`	math.h	`double fabs(double);`
`fclose*`	stdio.h	`int fclose(FILE *);`
`fcloseall*`	stdio.h	`int fcloseall(void);`
`feof*`	stdio.h	`int feof(FILE *);`
`fflush*`	stdio.h	`int fflush(FILE *);`
`fgetc*`	stdio.h	`int fgetc(FILE *);`
`fgetpos`	stdio.h	`int fgetpos(FILE *, fpos t *)`
`fgets*`	stdio.h	`char *fgets(char *, int, FILE *);`
`floor*`	math.h	`double floor(double);`
`flushall*`	stdio.h	`int flushall(void);`
`fmod*`	math.h	`double fmod(double, double);`
`fopen*`	stdio.h	`FILE *fopen(const char *, const char *);`
`fprintf*`	stdio.h	`int fprintf(FILE *, const char *, ...);`
`fputc*`	stdio.h	`int fputc(int, FILE *);`
`fputs*`	stdio.h	`int fputs(const char *, FILE *)`
`fread*`	stdio.h	`size t fread(void *, size t, size t, FILE *);`
`free*`	stdlib.h	`void free(void *);`
`freopen`	stdio.h	`FILE *freopen(const char *, const char *, FILE *);`
`frexp*`	math.h	`double frexp(double, int *);`
`fscanf*`	stdio.h	`int fscanf(FILE *, const char *, ...);`
`fseek*`	stdio.h	`int fseek(FILE *, long, int);`
`fsetpos`	stdio.h	`int fsetpos(FILE *, const fpos t *);`
`ftell*`	stdio.h	`long ftell(FILE *);`
`fwrite*`	stdio.h	`size t fwrite(const void *,size t, size t, FILE *);`
`getenv`	stdlib.h	`char *getenv(const char *);`

Tableau E.1 : Fonctions C courantes listées par ordre alphabétique *(suite)*

Fonction	*Fichier d'en-tête*	*Prototype de la fonction*
gets*	stdio.h	char *gets(char *); (à proscrire – utiliser fgets())
gmtime	time.h	struct tm *gmtime(const time t *);
isalnum*	ctype.h	int isalnum(int);
isalpha*	ctype.h	int isalpha(int);
isascii*	ctype.h	int isascii(int);
iscntrl*	ctype.h	int iscntrl(int);
isdigit*	ctype.h	int isdigit(int);
isgraph*	ctype.h	int isgraph(int);
islower*	ctype.h	int islower(int);
isprint*	ctype.h	int isprint(int);
ispunct*	ctype.h	int ispunct(int);
isspace*	ctype.h	int isspace(int);
isupper*	ctype.h	int isupper(int);
isxdigit*	ctype.h	int isxdigit(int);
labs	stdlib.h	long int labs(long int);
ldexp	math.h	double ldexp(double, int);
ldiv	stdlib.h	ldiv t div(long int, long int);
localtime*	time.h	struct tm *localtime(const time t *);
log*	math.h	double log(double);
log10*	math.h	double log10(double);
malloc*	stdlib.h	void *malloc(size t);
mblen	stdlib.h	int mblen(const char *, size t);
mbstowcs	stdlib.h	size t mbstowcs(wchar t *, const char *, size t);
mbtowc	stdlib.h	int mbtowc(wchar t *, const char *, size t);
memchr	string.h	void *memchr(const void *, int, size t);

Tableau E.1 : Fonctions C courantes listées par ordre alphabétique *(suite)*

Fonction	*Fichier d'en-tête*	*Prototype de la fonction*
`memcmp`	string.h	`int memcmp(const void *, const void *, size t);`
`memcpy`	string.h	`void *memcpy(void *, const void *, size t);`
`memmove`	string.h	`void *memmove(void *, const void *, size t);`
`memset`	string.h	`void *memset(void *, int, size t);`
`mktime*`	time.h	`time t mktime(struct tm *);`
`modf`	math.h	`double modf(double, double *);`
`perror*`	stdio.h	`void perror(const char *);`
`pow*`	math.h	`double pow(double, double);`
`printf*`	stdio.h	`int printf(const char *,...);`
`putc*`	stdio.h	`int putc(int, FILE *);`
`putchar*`	stdio.h	`int putchar(int);`
`puts*`	stdio.h	`int puts(const char *);`
`qsort*`	stdlib.h	`void qsort(void*, size t, size t, int (*)  (const void*, const void *));`
`rand`	stdlib.h	`int rand(void);`
`realloc*`	stdlib.h	`void *realloc(void *, size t);`
`remove*`	stdio.h	`int remove(const char *);`
`rename*`	stdio.h	`int rename(const char *, const char *);`
`rewind*`	stdio.h	`void rewind(FILE *);`
`scanf*`	stdio.h	`int scanf(const char *,...);`
`setbuf`	stdio.h	`void Setbuf(FILE *, char *);`
`setvbuf`	stdio.h	`int setvbuf(FILE *, char *, int, size t);`
`sin*`	math.h	`double sin(double);`
`sinh*`	math.h	`double sinh(double);`
`sleep*`	time.h	`void sleep(time t);`
`sprintf`	stdio.h	`int sprintf(char *, const char *, ...);`

Tableau E.1 : Fonctions C courantes listées par ordre alphabétique *(suite)*

Fonction	*Fichier d'en-tête*	*Prototype de la fonction*
`sqrt*`	math.h	`double sqrt(double);`
`srand`	stdlib.h	`void srand(unsigned);`
`sscanf`	stdio.h	`int sscanf(const char *, const char *, ...);`
`strcat*`	string.h	`char *strcat(char *, const char *);`
`strchr*`	string.h	`char *strchr(const char *, int);`
`strdup*`	string.h	`char *strdup(const char *);`
`strerror`	string.h	`char *strerror(int);`
`strftime*`	time.h	`size t strftime(char *,size t, const char *, const struct tm *);`
`strncat*`	string.h	`char *strncat(char *, const char *, size t);`
`strncmp*`	string.h	`int strncmp(const char *, const char *, size t);`
`strpbrk*`	string.h	`char *strpbrk(const char *, const char *);`
`strrchr*`	string.h	`char *strrchr(const char *, int);`
`strspn*`	string.h	`size t strspn(const char *, const char*);`
`strstr*`	string.h	`char *strstr(const char *, const char*);`
`strtod`	stdlib.h	`double strtod(const char *, char **);`
`strtok`	string.h	`char *strtok(char *, const char *);`
`strtol`	stdlib.h	`long strtol(const char *, char**, int);`
`strtoul`	stdlib.h	`unsigned long strtoul(const char*, char **, int);`
`strupr*`	string.h	`char *strupr(char *);`
`system*`	stdlib.h	`int system(const char *);`
`tan*`	math.h	`double tan(double);`
`tanh*`	math.h	`double tanh(double);`
`time*`	time.h	`time t time(time t *);`
`tmpfile`	stdio.h	`FILE *tmpfile(void);`
`tmpnam*`	stdio.h	`char *tmpnam(char *);` (préférer `tmpfile()`)

Tableau E.1 : Fonctions C courantes listées par ordre alphabétique *(suite)*

Fonction	*Fichier d'en-tête*	*Prototype de la fonction*
`tolower`	ctype.h	`int tolower(int);`
`toupper`	ctype.h	`int toupper(int);`
`ungetc*`	stdio.h	`int ungetc(int, FILE *);`
`va arg*`	stdarg.h	`(type) va arg(va list, (type));`
`va end*`	stdarg.h	`void va end(va list);`
`va start*`	stdarg.h	`void va start(va list, lastfix);`
`vfprintf`	stdio.h	`int vfprintf(FILE*, const char *, ...);`
`vprintf`	stdio.h	`int vprintf(const char *, ...);`
`vsprintf`	stdio.h	`int vsprintf(char *, const char *, ...);`
`wcstombs`	stdlib.h	`size t wcstombs(char *, const, size t);`
`wctomb`	stdlib.h	`int wctomb(char *, wchar t);`



Bibliothèques de fonctions

Le langage C est un langage puissant mais, du fait de son faible nombre d'instructions, il ne le montre pas au premier abord. Pourtant, à l'aide de bibliothèques de fonctions, vous pouvez ouvrir une fenêtre, chercher une page Web ou interroger une base de données en quelques lignes.

Les bibliothèques de fonctions

Une bibliothèque de fonctions est un ensemble de fonctions que vous pouvez utiliser comme des fonctions standard du C. Pour l’utiliser, vous devez l’avoir installée au préalable sur votre système. Reportez-vous au mode d’emploi de votre système ou de la bibliothèque pour savoir comment l’installer.

Une bibliothèque est composée de deux parties. L’une regroupe les fonctions, directement utilisables par votre programme compilé. C’est elle qu’on appelle bibliothèque au sens commun du terme. Elle porte généralement une extension *.a* pour sa version statique, *.so* pour sa version dynamique sur Unix et *.dll* pour sa version dynamique sur Windows. Elle est dite dynamique car elle est chargée dynamiquement sur le système lorsqu’un programme en a besoin. Elle est également appelée bibliothèque partagée dans ce cas. L’autre partie d’une bibliothèque consiste en les fichiers d’en-tête, nécessaires uniquement lors de la compilation.

Pour compiler un programme faisant appel à une bibliothèque externe, vous n’avez qu’à indiquer les fichiers d’en-têtes de cette bibliothèque avec `#include`. Les choses changent pour l’édition des liens. En effet, vous devez indiquer où se trouve la bibliothèque à l’éditeur de liens. Si vous choisissez d’intégrer la version statique de la bibliothèque à votre programme, indiquez simplement le nom de la bibliothèque dont l’extension est *.a* comme vous auriez indiqué un fichier objet d’extension *.o*. Cela présente l’avantage que votre exécutable ne nécessite plus la bibliothèque comme dépendance. En contrepartie, votre exécutable sera d’autant plus gros en mémoire et sur le disque. Pour cette raison, la version statique des bibliothèques est peu utilisée de nos jours. Pour utiliser la version dynamique, voyez le manuel de votre éditeur de lien. Par exemple, les compilateurs CC sur Unix et GCC nécessitent l’option `L` pour indiquer le répertoire dans lequel se trouve la bibliothèque et `l` pour le nom de la bibliothèque. Ce nom est le nom du fichier sans son extension et sans le préfixe *lib*. Voici ce que cela donne pour compiler *monprog* et le lier à la bibliothèque *libmysql.so* :

```
gcc -g monprog.c
gcc -lmysql -o monprog monprog.o
```

La première ligne compile *monprog.c* et génère l’objet *monprog.o*. La seconde réalise l’édition des liens entre les divers objets (*monprog.o* dans notre exemple) et les bibliothèques partagées (ici *libmysql.so*, dont on enlève le préfixe *lib* et l’extension *.so*). Le résultat est l’exécutable dont le nom du fichier est indiqué avec `o`, à savoir *monprog*.

Dans la suite de ces annexes, nous vous présentons diverses bibliothèques que vous pouvez utiliser dans vos programmes. Vous prêterez une attention particulière à deux points : la portabilité de ces bibliothèques et surtout la licence de celles-ci. Au niveau de la portabilité, nous n’avons vérifié que pour Linux (et *a priori* certains Unix), Windows et

Mac OS X. Certaines bibliothèques sont également portées sur de nombreux autres systèmes. La licence est aussi très importante car elle peut avoir une incidence sur la façon dont vous pourrez distribuer votre programme, le donner ou le vendre. Ainsi, une particularité de la licence GPL (toutes versions) est que, si vous liez votre programme à une bibliothèque sous cette licence, celui-ci devra alors lui aussi être distribué selon les termes de cette licence GPL. Cela peut paraître contraignant. C'est pourtant grâce à cela que les logiciels libres sont si nombreux aujourd'hui. Voyez le site **http://www.gnu.org/licenses/license-list.html**, qui présente un résumé de diverses licences libres et ce que vous pouvez faire avec.



Nous n'avons testé personnellement que certaines des bibliothèques présentées dans cette annexe. Les autres bibliothèques ont été recensées selon la documentation de leur site Web respectif et selon leur popularité.

Structures de données

Nous avons regroupé ci-après les données relatives au traitement des données, que ce soit leur représentation en mémoire ou sur disque avec glib et libXML2, le chiffrage et déchiffrage ou la compression et la décompression.

Bibliothèques pour traiter et gérer les données

Bibliothèque	*Licence*	*Compatibilité*	*URL*	*Remarques*
Glib	LGPL	Unix/Linux, Windows, Mac OS X	**http://www.gtk.org/**	Traitement des listes chaînées, tables de hachage, arbres et autres fonctions utiles
openssl	Dérivée de la licence Apache	Unix/Linux, Windows, Mac OS X	**http://www.openssl.org/**	Encapsulation chiffrée de données pour des transferts sécurisés
gnutls	LGPL	Unix/Linux, Windows, Mac OS X	**http://www.gnu.org/software/gnutls/**	Encapsulation chiffrée de données pour des transferts sécurisés
NSS	MPL/GPL/LGPL	Unix/Linux, Mac OS X	**http://www.mozilla.org/projects/security/pki/nss/**	Encapsulation chiffrée de données pour des transferts sécurisés
libXML2	MIT	Unix/Linux, Windows, Mac OS X	**http://xmlsoft.org/**	Analyse et traitement des données en XML

Bibliothèques pour traiter et gérer les données *(suite)*

Bibliothèque	*Licence*	*Compatibilité*	*URL*	*Remarques*
zlib	libre	Unix/Linux, Windows, Mac OS X	**http://zlib.net/**	Compression et décompression des données gzip
libbzip2	libre	Unix/Linux, Windows, Mac OS X	**http://www.bzip.org/**	Compression et décompression des données bzip2

Interfaces utilisateur

La construction d'une interface utilisateur passe obligatoirement par une bibliothèque graphique à moins de vouloir réinventer le monde. Chaque système ayant son interface utilisateur dédiée, ces bibliothèques ne sont historiquement pas portables. L'apparition des bibliothèques libres sur Linux comme GTK+ (et de QT pour les programmeurs en C++) a fait bouger les choses. Les plus portables sont aujourd'hui GTK+ et Wxwidgets. Lesstif correspond à un ancien standard d'interfaces graphiques qui s'appelait MOTIF et ne se développe plus guère. Nous l'avons laissé dans le tableau en guise de souvenir.

Bibliothèques pour les interfaces utilisateur

Bibliothèque	*Licence*	*Compatibilité*	*URL*	*Remarques*
GTK+	LGPL	Unix/Linux, Windows, Mac OS X	**http://www.gtk.org/**	Bibliothèque des projets Gimp et GNOME, entre autres
Libglade	LGPL	Unix/Linux, Windows, Mac OS X	**http://glade.gnome.org/**	Charge les interfaces dessinées avec Glade, qui se base sur GTK+
Lesstif	LGPL	Unix/Linux, Windows, Mac OS X	**http://www.lesstif.org/**	Copie libre de la bibliothèque MOTIF
Wxwidgets	Licence wxWindows	Unix/Linux, Windows, Mac OS X	**http://www.wxwidgets.org/**	Boîte à outils graphiques dont le but est d'implé-menter un support natif des interfaces graphiques de diverses plates-formes

Bibliothèques pour les interfaces utilisateur *(suite)*

Bibliothèque	*Licence*	*Compatibilité*	*URL*	*Remarques*
MFC (Microsoft)	Propriétaire	Windows	**http://msdn.microsoft.com/**	Interface graphique native des systèmes Windows
Ncurses	MIT	Unix/Linux, Windows, Mac OS X	**http://www.gnu.org/software/ncurses/ncurses.html**	Émulation de curses (interface en mode texte de System V release 4)

Jeux et multimédia

Les jeux et le multimédia ne sont pas en reste et certaines bibliothèques devraient vous faciliter les choses. Nous avons cité peu de bibliothèques dans le tableau suivant car, pour les jeux, les bibliothèques sont souvent issues d'un groupe de programmeurs et dédiées aux jeux de ce groupe. Rares sont celles qui, comme SDL, ont réussi à devenir plus génériques.

Bibliothèques pour les jeux et le multimédia

Bibliothèque	*Licence*	*Compatibilité*	*URL*	*Remarques*
SDL	GPLv2	Unix/Linux, Windows, Mac OS X	**http://www.libsdl.org/**	Bibliothèque multimédia (audio, clavier, souris, joystick, 2D, 3D…)
allegro	libre	Unix/Linux, Windows, Mac OS X	**http://alleg.sourceforge.net/**	Bibliothèque multimédia (audio, clavier, souris, joystick, 2D, 3D…)
libjpeg	libre	Unix/Linux, Windows, Mac OS X	**http://www.ijg.org/**	Graphisme au format JPEG
libpng	libre	Unix/Linux, Windows, Mac OS X	**http://www.libpng.org/pub/png/libpng.html**	Graphisme au format PNG

Programmation réseau

La programmation réseau est un domaine de prédilection du C et il est possible de faire communiquer des programmes entre eux sans faire appel à des bibliothèques de fonctions externes. Voyez à ce sujet la documentation des fonctions `socket()`, `bind()`, `listen()` et `accept()`. Néanmoins, dès qu'il s'agit de communiquer selon un protocole comme HTTP, FTP, POP3, SOAP ou Jabber, il est préférable d'utiliser une bibliothèque qui vous facilitera la programmation. Par exemple, vous pouvez télécharger une page Web en quelques lignes de code avec libcurl !

Bibliothèques pour la programmation réseau

Bibliothèque	*Protocole*	*Licence*	*Compatibilité*	*URL*	*Objet du protocole*
libcurl	HTTP, HTTPS	Dérivée de la licence MIT	Unix/Linux, Windows, Mac OS X	**http://curl.haxx.se/libcurl/**	Transfert de données sur le World Wide Web
libcurl	FTP	Dérivée de la licence MIT	Unix/Linux, Windows, Mac OS X	**http://curl.haxx.se/libcurl/**	Transfert de fichiers
GNU mailutils	IMAP, POP3	GNU LGPL	Unix/Linux	**http://www.gnu.org/software/mailutils/**	Lecture de boîtes aux lettres distantes
UW c-client	IMAP, POP3	Apache licence version 2.0	Unix/Linux, Windows, Mac OS X	**http://www.washington.edu/imap/**	Lecture de boîtes aux lettres distantes
GNU mailutils	SMTP	GNU LGPL	Unix/Linux	**http://www.gnu.org/software/mailutils/**	Envoi de messages électroniques
UW c-client	SMTP	Apache licence version 2.0	Unix/Linux, Windows, Mac OS X	**http://www.washington.edu/imap/**	Envoi de messages électroniques
gsoap	SOAP	Dérivée de Mozilla Public Licence 1.1 ou GPL ou commerciale	Unix/Linux, Windows, Mac OS X	**http://gsoap2.sourceforge.net/**	Transmission de messages entre objets distants (RPC) au format XML *via* HTTP ou SMTP

Bibliothèques pour la programmation réseau *(suite)*

Bibliothèque	*Protocole*	*Licence*	*Compatibilité*	*URL*	*Objet du protocole*
Neon	WebDAV	GPLv2	Unix/Linux, Windows, Mac OS X	**http://webdav.org/neon/**	Transfert simplifié de fichiers entre serveurs distants
loudmouth	Jabber	LGPL	Unix/Linux, Windows, Mac OS X	**http://www.loudmouth-project.org/**	Technologie de flux XML utilisée principalement pour la messagerie instantanée
avahi	DNS Service Discovery	LGPL	Unix/Linux	**http://www.avahi.org/**	Découverte de services dans un réseau local

Bases de données et annuaires

Chaque moteur de base de données digne de ce nom est fourni avec une bibliothèque qui permet d'effectuer des requêtes à partir de fonctions C. Il n'en existe généralement qu'une par moteur de base de données. Il existe néanmoins deux exceptions notables : freetds pour se connecter aux serveurs Sybase et Microsoft SQL Server et, surtout, les implémentations ODBC (unixodbc sur Unix et les MFC sur Windows), qui permettent d'accéder à toutes les bases disposant d'un pilote ODBC.

Dédiés également au stockage de données mais sous une forme différente, les annuaires sont surtout représentés par LDAP et ses diverses implémentations. Nous vous présentons ici deux bibliothèques concurrentes et libres.

Bibliothèques pour les bases de données et annuaires

Bibliothèque	*Base/ annuaire*	*Licence*	*Compatibilité*	*URL*
Libmysql	MySQL	GPL	Unix/Linux, Windows, Mac OS X	**http://www.mysql.org/**
Libpq	Postgresql	BSD	Unix/Linux, Windows, Mac OS X	**http://www.postgresql.org**

Bibliothèques pour les bases de données et annuaires *(suite)*

Bibliothèque	*Base/ annuaire*	*Licence*	*Compatibilité*	*URL*
freetds	Sybase et Microsoft SQL Server	LGPL	Unix/Linux, Windows, Mac OS X	**http://freetds.org/**
Unixodbc	ODBC	LGPL	Unix/Linux, Mac OS X	**http://www.unixodbc.org/**
Microsoft Foundation Classes (MFC)	ODBC	Propriétaire	Windows	**http://msdn.microsoft.com/**
Fedora Directory Server	LDAP	MPL/LGPL/ GPL	Unix/Linux	**http:// directory.fedora.redhat.com/**
openldap	LDAP	Open LDAP	Unix/Linux, Windows, Mac OS X	**http://www.openldap.org/**

G

Les Logiciels libres

Lorsqu'un logiciel est diffusé, il est souvent soumis à licence. Afin de pouvoir utiliser ce logiciel, vous devez accepter celle-ci. Qu'est-ce qu'une licence, au juste ? À quoi sert-elle ? En quoi allez-vous être concerné, vous qui programmez en C et peut-être d'autres langages ?

Licence et copyright

Un logiciel est une œuvre de l'esprit dont les droits appartiennent à l'auteur. En droit français, ces droits sont d'une part le droit moral et d'autre part les droits patrimoniaux. Le droit moral est un droit de regard sur l'œuvre qui est inaliénable. Les droits patrimoniaux, autrement appelés *copyright*, comprennent entre autres les droits d'exploitation. Les droits patrimoniaux peuvent être cédés, généralement moyennant finance.

Celui qui dispose des droits patrimoniaux ou, en droit anglo-saxon, du *copyright* peut utiliser et surtout diffuser un logiciel. Si vous n'avez pas le copyright sur un logiciel, vous n'avez pas le droit de faire quoi que ce soit avec, pas même de l'utiliser. Mais, comme la tondeuse à gazon du voisin, qu'il ne vous viendrait pas à l'esprit d'utiliser sans sa permission, vous pouvez demander au détendeur du copyright le droit d'utiliser son logiciel. Vous obtiendrez généralement ce droit *via* la licence, qui peut être considérée comme un contrat entre vous deux.

La licence est une définition des droits que le détendeur du copyright cède aux utilisateurs. Seul celui qui a le copyright peut la choisir et la changer. Elle fixe entre autres les règles en matière d'utilisation et de diffusion du logiciel. Par exemple, vous pourrez utiliser certains logiciels dits propriétaires à condition de payer une certaine somme, pendant une durée fixée, et ne pourrez pas diffuser vous-même le logiciel.

En tant qu'auteur des logiciels que vous ne manquerez pas d'écrire suite à la lecture de ce livre, vous allez disposer du copyright sur vos œuvres. Vous allez donc pouvoir diffuser vos réalisations. Pour que vos amis, vos clients ou toute autre personne puisse utiliser vos créations, vous devrez leur indiquer dans quel cadre ils peuvent les utiliser. Vous écrirez donc une licence qu'ils devront accepter.

Avant d'aborder la suite, vous remarquerez que, si vous ne souhaitez pas diffuser vos créations, elles seront naturellement protégées par le copyright et vous n'aurez pas besoin de vous préoccuper d'une licence. C'est d'ailleurs le cas de la grande majorité des logiciels, qui sont écrits en entreprises pour elles-mêmes et n'ont pas vocation à être diffusés, encore moins à l'entreprise concurrente !

Qu'est-ce qu'un logiciel libre ?

Un logiciel libre est un logiciel dont la licence a quatre caractéristiques que l'on appelle parfois les quatre libertés fondamentales d'un logiciel. Les voici :

- la liberté d'utiliser le logiciel dans n'importe quel but ;
- la liberté d'étudier comment fonctionne le logiciel et de l'adapter à ses besoins ;

- la liberté de redistribuer des copies du logiciel ;
- la liberté d'améliorer le logiciel et de diffuser ces améliorations pour que tous puissent en bénéficier.

Vous remarquez que deux de ces libertés impliquent que le code source du logiciel soit accessible. On dit par ailleurs qu'un logiciel est propriétaire dès lors qu'une de ces quatre libertés n'est pas respectée.

Les logiciels libres ont leurs détracteurs car certains voient en eux le vol de la propriété intellectuelle. C'est une erreur de croire cela car il s'agit non pas d'un vol mais au contraire d'un don. C'est la même différence qu'il y a, lorsque vous offrez un bouquet de fleurs, entre imposer de le mettre dans un vase transparent et, au contraire, laisser la liberté de choisir le vase, voire de le mettre ailleurs que dans un vase.

D'autres ont peur de diffuser leur logiciel sous une licence libre par crainte que ses fonctionnalités, en particulier celles innovantes, soient reprises dans des logiciels concurrents. Si vous avez cette crainte, demandez-vous si la concurrence vous fait peur et pourquoi. C'est peut-être un premier signe de faiblesse pour votre logiciel. Au contraire, si la concurrence reprend vos fonctionnalités, soyez-en fier : elle vous fait de la publicité ! D'ailleurs, si les logiciels libres étaient moins bons que leurs concurrents propriétaires, ils auraient déjà disparu. Pourtant, cela fait plus de vingt ans qu'ils défient les autres, qu'ils soient propriétaires ou également libres, et gagnent, pour certains, des parts de marché non négligeables, comme leur emblème, GNU/Linux.

D'autres encore imaginent que, comme ils donnent le droit de redistribuer des copies du logiciel, logiciels libres et logiciels commerciaux sont incompatibles. Cela n'est pas aussi évident et, pendant de nombreuses années, les logiciels libres étaient tous diffusés gratuitement. Mais rien ne vous empêche de vendre un logiciel libre. Cela est garanti par la première des libertés : votre but peut être de gagner de l'argent. Mieux, vous pouvez vendre un logiciel libre dont vous n'avez pas le copyright. Cela est garanti par la troisième liberté, qui ne précise pas dans quelles conditions redistribuer les copies du logiciel ! Pour que ce modèle économique fonctionne, vous devez en revanche vendre une prestation autour du logiciel, par exemple un service d'installation, une garantie de fonctionnement ou un travail d'intégration de divers logiciels libres pour faciliter leur installation. C'est par exemple tout cela que la société Redhat propose à ses clients.

Enfin, si vous développez un petit programme, interrogez-vous si vous tirerez plus de bénéfice à gagner un peu d'argent en le vendant à quelques utilisateurs honnêtes ou à bénéficier d'améliorations que d'autres ne manqueront pas de vous proposer, ayant eu accès libre au code source. Certains indiquent d'ailleurs que, bien que leur logiciel soit libre, ils ne sont pas contre un peu d'argent.

Différences entre les diverses licences

Écrire une licence pour un logiciel n'est pas forcément très simple. Mieux vaut avoir un juriste parmi ses amis ou dans sa famille. Si vous souhaitez que votre logiciel soit libre, il existe des modèles de licences, ou licences dans le langage courant. Vous pouvez mettre votre logiciel sous l'une d'elles. Parmi les plus connues, voici un tableau avec leur nom, l'endroit où vous pouvez lire leur texte intégral et des exemples de logiciels connus protégés par elles.

Licences

Nom	***URL du texte intégral***	***Exemple***
GPL v2	**http://www.gnu.org/licenses/old-licenses/gpl-2.0.html**	Linux-2.6
GPL v3	**http://www.gnu.org/licenses/gpl.html**	GCC
LGPL v2	**http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html**	GTK+
LGPL v3	**http://www.gnu.org/licenses/lgpl.html**	Gnustep
FreeBSD	**http://www.freebsd.org/copyright/freebsd-license.html**	FreeBSD
MPL	**http://www.mozilla.org/MPL/MPL-1.1.html**	Mozilla Firefox
Freeware	N/A	Logiciels gratuits sans accès au code source
Shareware	N/A	Logiciels dont la gratuité est limitée dans le temps et sans accès au code source
Domaine public	Le copyright a été abandonné par son auteur	

Nous avons inclus dans le tableau les *freewares*, *sharewares* et les logiciels du domaine public. Ces trois sortes de logiciels ne doivent pas être confondus avec les logiciels libres malgré leur gratuité. Ces trois catégories de logiciels ont pour point commun de ne pas diffuser le code source ou, s'ils le diffusent, de ne pas permettre explicitement de le modifier ou de le réutiliser.

Pour choisir une licence, vous pouvez lire le texte de chacune d'elles. Si cela vous semble trop rébarbatif, il existe des résumés de ce qu'elles permettent et interdisent. Sachez néanmoins que la caractéristique la plus remarquable de la licence GPL (toutes versions) est ce qu'on appelle le copyleft. Cela consiste à imposer que tout ajout ou modification sur le code d'un logiciel est également soumis à cette même licence GPL. De même, si vous réutilisez du code d'un logiciel placé sous licence GPL, ce code, qui reste sous licence

GPL, impose que tout votre programme soit également sous GPL. Cela peut parfois vous imposer le choix de la licence. C'est ce fameux copyleft qui a répandu la GPL sur de nombreux logiciels et a participé à l'essor des logiciels libres. Vous pouvez voir dans le copyleft une diminution de liberté. Si cela vous gêne, optez plutôt pour une licence FreeBSD. Mais le copyleft peut également vous garantir que vous aurez accès à toutes les modifications de votre code qui seraient diffusées.

Diffuser un logiciel libre

Si vous diffusez un de vos logiciels, vous devez impérativement indiquer que c'est vous qui avez le copyright, en écrivant dans une notice jointe une mention avec votre nom et la date, comme celle-ci :

```
Copyright 2008 Pierre Martin
```

N'utilisez ni symbole © ni l'équivalent `(c)` mais toujours le mot anglais `Copyright` par convention internationale.

Vous devez également indiquer dans quels termes vous diffusez votre logiciel, autrement dit une licence. Dans le cas d'une licence libre, indiquez dans la même notice le nom de la licence et le nom du fichier qui contient le texte intégral de la licence, par exemple *COPYING*. Dans *COPYING*, collez le texte que vous aurez copié du site où se trouve ce texte intégral. Indiquez également le copyright et le nom de la licence dans chacun des fichiers qui composent votre code source.

Si vous avez choisi la licence GPL, il est conseillé de placer une copie verbatim du texte suivant au début de chacun de vos fichiers sources :

```
/*
 *   This file is part of MonProgramme.
 *
 *   Foobar is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   Foobar is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with Foobar; if not, write to the Free Software
 *   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 */
```

Vous pouvez trouver ce texte ainsi que quelques conseils sur le site de GNU à l'adresse **http://www.gnu.org/licenses/gpl-howto.fr.html**.

Traduction française de la licence GPL version 2

Ceci est une traduction non officielle de la *GNU General Public License* version 2 en français. Elle n'a pas été publiée par la Free Software Foundation et ne détermine pas les termes de distribution pour les logiciels qui utilisent la GNU GPL ; seul le texte anglais original de la GNU GPL détermine ces termes. Nous vous proposons néanmoins le texte de la version 2 de cette licence en guise d'exemple de licence pour un logiciel libre.

La traduction suivante est issue du site de la Free Software Foundation France, dont la page d'accueil est à l'adresse **http://fsffrance.org**. Nous ne donnons pas l'adresse de la traduction car, avec l'apparition récente de la version 3, elle risque de changer d'adresse rapidement. Vous pourrez la retrouver en effectuant une recherche sur ce site ou sur celui de GNU (**http://www.gnu.org**).

Licence publique générale GNU

Les licences de la plupart des logiciels sont conçues pour vous enlever toute liberté de les partager et de les modifier.

A contrario, la Licence publique générale est destinée à garantir votre liberté de partager et de modifier les logiciels libres et à assurer que ces logiciels soient libres pour tous leurs utilisateurs.

La présente Licence publique générale s'applique à la plupart des logiciels de la Free Software Foundation, ainsi qu'à tout autre programme pour lequel ses auteurs s'engagent à l'utiliser.

(Certains autres logiciels de la Free Software Foundation sont couverts par la GNU Lesser General Public License à la place.)

Vous pouvez aussi l'appliquer aux programmes qui sont les vôtres.

Quand nous parlons de logiciels libres, nous parlons de liberté, non de prix.

Nos licences publiques générales sont conçues pour vous donner l'assurance d'être libre de distribuer des copies des logiciels libres (et de facturer ce service, si vous le souhaitez), de recevoir le code source ou de pouvoir l'obtenir si vous le souhaitez, de pouvoir modifier les logiciels ou en utiliser des éléments dans de nouveaux programmes libres et de savoir que vous pouvez le faire.

Pour protéger vos droits, il nous est nécessaire d'imposer des limitations qui interdisent à quiconque de vous refuser ces droits ou de vous demander d'y renoncer.

Certaines responsabilités vous incombent en raison de ces limitations si vous distribuez des copies de ces logiciels ou si vous les modifiez.

Par exemple, si vous distribuez des copies d'un tel programme, à titre gratuit ou contre une rémunération, vous devez accorder aux destinataires tous les droits dont vous disposez.

Vous devez vous assurer qu'eux aussi reçoivent le code source ou puissent en disposer.

Et vous devez leur montrer les présentes conditions afin qu'ils aient connaissance de leurs droits.

Nous protégeons vos droits en deux étapes : (1) nous sommes titulaires des droits d'auteur du logiciel ; (2) nous vous délivrons cette licence, qui vous donne l'autorisation légale de copier, de distribuer et/ou de modifier le logiciel.

En outre, pour la protection de chaque auteur ainsi que la nôtre, nous voulons nous assurer que chacun comprenne que ce logiciel libre ne fait l'objet d'aucune garantie.

Si le logiciel est modifié par quelqu'un d'autre puis transmis à des tiers, nous voulons que les destinataires soient mis au courant que ce qu'ils ont reçu n'est pas le logiciel d'origine, de sorte que tout problème introduit par d'autres ne puisse entacher la réputation de l'auteur originel.

En définitive, un programme libre restera à la merci des brevets de logiciels.

Nous souhaitons éviter le risque que les redistributeurs d'un programme libre fassent des demandes individuelles de licence de brevet, ceci ayant pour effet de rendre le programme propriétaire.

Pour éviter cela, nous établissons clairement que toute licence de brevet doit être concédée de façon que l'usage en soit libre pour tous ou bien qu'aucune licence ne soit concédée.

Les termes exacts et les conditions de copie, de distribution et de modification sont les suivants.

Conditions de copie, de distribution et de modification de la Licence publique générale GNU

0. La présente Licence s'applique à tout programme ou à tout autre ouvrage contenant un avis, apposé par le titulaire des droits d'auteur, stipulant qu'il peut être distribué au titre des conditions de la présente Licence publique générale.

Ci-après, le "Programme" désigne l'un quelconque de ces programmes ou ouvrages, et un "ouvrage fondé sur le Programme" désigne soit le Programme, soit un ouvrage qui en dérive au titre des lois sur le droit d'auteur : en d'autres termes, un ouvrage contenant le Programme ou une partie de ce dernier, soit à l'identique, soit avec des modifications et/ou traduit dans un autre langage.

(Ci-après, le terme "modification" implique, sans s'y réduire, le terme traduction.)

Chaque concessionnaire sera désigné par "vous".

Les activités autres que la copie, la distribution et la modification ne sont pas couvertes par la présente Licence ; elles sont hors de son champ d'application.

L'opération consistant à exécuter le Programme n'est soumise à aucune limitation et les sorties du Programme ne sont couvertes que si leur contenu constitue un ouvrage fondé sur le Programme (indépendamment du fait qu'il ait été réalisé par l'exécution du Programme).

La validité de ce qui précède dépend de ce que fait le Programme.

1. Vous pouvez copier et distribuer des copies à l'identique du code source du Programme tel que vous l'avez reçu, sur n'importe quel support, du moment que vous apposiez sur chaque copie, de manière *ad hoc* et parfaitement visible, l'avis de droit d'auteur adéquat et une exonération de garantie ; que vous gardiez intacts tous les avis faisant référence à la présente Licence et à l'absence de toute garantie ; et que vous fournissiez à tout destinataire du Programme autre que vous-même un exemplaire de la présente Licence en même temps que le Programme.

Vous pouvez faire payer l'acte physique de transmission d'une copie, et vous pouvez, à votre discrétion, proposer une garantie contre rémunération.

2. Vous pouvez modifier votre copie ou des copies du Programme ou n'importe quelle partie de celui-ci, créant ainsi un ouvrage fondé sur le Programme, et copier et distribuer de telles modifications ou un tel ouvrage selon les termes de l'Article 1 ci-dessus, à condition de vous conformer également à chacune des obligations suivantes :

a) Vous devez munir les fichiers modifiés d'avis bien visibles stipulant que vous avez modifié ces fichiers, ainsi que la date de chaque modification.

b) Vous devez prendre les dispositions nécessaires pour que tout ouvrage que vous distribuez ou publiez et qui, en totalité ou en partie, contient ou est fondé sur le Programme – ou une partie quelconque de ce dernier – soit concédé comme un tout, à titre gratuit, à n'importe quel tiers, au titre des conditions de la présente Licence.

c) Si le programme modifié lit habituellement des instructions de façon interactive lorsqu'on l'exécute, vous devez, quand il commence son exécution pour ladite utilisation interactive de la manière la plus usuelle, faire en sorte qu'il imprime ou affiche une annonce comprenant un avis de droit d'auteur *ad hoc* et un avis stipulant qu'il n'y a pas de garantie (ou bien indiquant que c'est vous qui fournissez la garantie) et que les utilisateurs peuvent redistribuer le programme en respectant les présentes obligations et en expliquant à l'utilisateur comment voir une copie de la présente Licence.

(Exception : si le Programme est lui-même interactif mais n'imprime pas habituellement une telle annonce, votre ouvrage fondé sur le Programme n'est pas obligé d'imprimer une annonce.)

Ces obligations s'appliquent à l'ouvrage modifié pris comme un tout.

Si des éléments identifiables de cet ouvrage ne sont pas fondés sur le Programme et peuvent raisonnablement être considérés comme des ouvrages indépendants distincts en eux-mêmes, alors, la présente Licence et ses conditions ne s'appliquent pas à ces éléments lorsque vous les distribuez en tant qu'ouvrages distincts.

Mais, lorsque vous distribuez ces mêmes éléments comme partie d'un tout, lequel constitue un ouvrage fondé sur le Programme, la distribution de ce tout doit être soumise aux conditions de la présente Licence, et les autorisations qu'elle octroie aux autres concessionnaires s'étendent à l'ensemble de l'ouvrage et par conséquent à chaque et à toute partie indifféremment de qui l'a écrite.

Par conséquent, l'objet du présent article n'est pas de revendiquer des droits ou de contester vos droits sur un ouvrage entièrement écrit par vous ; son objet est plutôt d'exercer le droit de contrôler la distribution d'ouvrages dérivés ou d'ouvrages collectifs fondés sur le Programme.

De plus, la simple proximité du Programme avec un autre ouvrage qui n'est pas fondé sur le Programme (ou un ouvrage fondé sur le Programme) sur une partition d'un espace de stockage ou un support de distribution ne place pas cet autre ouvrage dans le champ d'application de la présente Licence.

3. Vous pouvez copier et distribuer le Programme (ou un ouvrage fondé sur lui, selon l'Article 2) sous forme de code objet ou d'exécutable, selon les termes des Articles 1 et 2 ci-dessus, à condition que vous accomplissiez l'un des points suivants :

a) L'accompagner de l'intégralité du code source correspondant, sous une forme lisible par un ordinateur, lequel doit être distribué au titre des termes des Articles 1 et 2 ci-dessus, sur un support habituellement utilisé pour l'échange de logiciels.

b) Ou l'accompagner d'une proposition écrite, valable pendant au moins trois ans, de fournir à tout tiers, à un tarif qui ne soit pas supérieur à ce que vous coûte l'acte physique de réaliser une distribution source, une copie intégrale du code source correspondant sous une forme lisible par un ordinateur, qui sera distribuée au titre des termes des Articles 1 et 2 ci-dessus, sur un support habituellement utilisé pour l'échange de logiciels.

c) Ou l'accompagner des informations reçues par vous concernant la proposition de distribution du code source correspondant. (Cette solution n'est autorisée que dans le cas d'une distribution non commerciale et seulement si vous avez reçu le programme sous forme de code objet ou d'exécutable accompagné d'une telle proposition – en conformité avec le sous-Article b ci-dessus.)

Le code source d'un ouvrage désigne la forme favorite pour travailler à des modifications de cet ouvrage. Pour un ouvrage exécutable, le code source intégral désigne la totalité du code source de la totalité des modules qu'il contient, ainsi que les éventuels fichiers de définition des interfaces qui y sont associés, ainsi que les scripts utilisés pour contrôler la compilation et l'installation de l'exécutable. Cependant, par exception spéciale, le code source distribué n'est pas censé inclure quoi que ce soit de normalement distribué (que ce soit sous forme source ou binaire) avec les composants principaux (compilateur, noyau, et autre) du système d'exploitation sur lequel l'exécutable tourne, à moins que ce composant lui-même n'accompagne l'exécutable.

Si distribuer un exécutable ou un code objet consiste à offrir un accès permettant leur copie depuis un endroit particulier, alors, l'offre d'un accès équivalent pour copier le code source depuis le même endroit compte comme une distribution du code source – même si les tiers ne sont pas contraints de copier le source en même temps que le code objet.

4. Vous ne pouvez copier, modifier, concéder en sous-licence ou distribuer le Programme, sauf tel qu'expressément prévu par la présente Licence. Toute tentative de copier, de modifier, de concéder en sous-licence ou de distribuer le Programme d'une autre manière est réputée non valable et met immédiatement fin à vos droits au titre de la présente Licence. Toutefois, les tiers ayant reçu de vous des copies, ou des droits, au titre de la présente Licence ne verront pas leurs autorisations résiliées aussi longtemps que lesdits tiers se conforment pleinement à elle.

5. Vous n'êtes pas obligé d'accepter la présente Licence étant donné que vous ne l'avez pas signée. Cependant, rien d'autre ne vous accorde l'autorisation de modifier ou de distribuer le Programme ou les ouvrages fondés sur lui. Ces actions sont interdites par la loi si vous n'acceptez pas la présente Licence. En conséquence, en modifiant ou en distribuant le Programme (ou un ouvrage quelconque fondé sur le Programme), vous signifiez votre acceptation de la présente Licence en le faisant et de toutes ses conditions concernant la copie, la distribution ou la modification du Programme ou d'ouvrages fondés sur lui.

6. Chaque fois que vous redistribuez le Programme (ou n'importe quel ouvrage fondé sur le Programme), une licence est automatiquement concédée au destinataire par le concédant originel de la licence, l'autorisant à copier, à distribuer ou à modifier le Programme, sous réserve des présentes conditions. Vous ne pouvez imposer une quelconque limitation supplémentaire à l'exercice des droits octroyés au titre des présentes par le destinataire. Vous n'avez pas la responsabilité d'imposer le respect de la présente Licence à des tiers.

7. Si, conséquemment à une décision de justice ou à l'allégation d'une transgression de brevet ou pour toute autre raison (non limitée à un problème de brevet), des obligations vous sont imposées (que ce soit par jugement, conciliation ou autre) qui contredisent les conditions de la présente Licence, elles ne vous excusent pas des conditions de la présente Licence. Si vous ne pouvez distribuer de manière à satisfaire simultanément vos obligations

au titre de la présente Licence et toute autre obligation pertinente, alors, il en découle que vous ne pouvez pas du tout distribuer le Programme. Par exemple, si une licence de brevet ne permettait pas une redistribution sans redevance du Programme par tous ceux qui reçoivent une copie directement ou indirectement par votre intermédiaire, alors, la seule façon pour vous de satisfaire à la fois à la licence du brevet et à la présente Licence serait de vous abstenir totalement de toute distribution du Programme.

Si une partie quelconque de cet article est tenue pour nulle ou inopposable dans une circonstance particulière quelconque, l'intention est que le reste de l'article s'applique. La totalité de la section s'appliquera dans toutes les autres circonstances.

Cet article n'a pas pour but de vous induire à transgresser un quelconque brevet ou d'autres revendications à un droit de propriété ou à contester la validité de la moindre de ces revendications ; cet article a pour seul objectif de protéger l'intégrité du système de distribution du logiciel libre, qui est mis en œuvre par la pratique des licences publiques. De nombreuses personnes ont fait de généreuses contributions au large spectre de logiciels distribués par ce système en se fiant à l'application cohérente de ce système ; il appartient à chaque auteur/donateur de décider si il ou elle veut distribuer du logiciel par l'intermédiaire d'un quelconque autre système, et un concessionnaire ne peut imposer ce choix.

Cet article a pour but de rendre totalement limpide ce que l'on pense être une conséquence du reste de la présente Licence.

8. Si la distribution et/ou l'utilisation du Programme est limitée dans certains pays que ce soit par des brevets ou par des interfaces soumises au droit d'auteur, le titulaire originel des droits d'auteur qui décide de couvrir le Programme par la présente Licence peut ajouter une limitation géographique de distribution explicite qui exclut ces pays afin que la distribution soit permise seulement dans ou entre les pays qui ne sont pas ainsi exclus. Dans ce cas, la présente Licence incorpore la limitation comme si elle était écrite dans le corps de la présente Licence.

9. La Free Software Foundation peut, de temps à autre, publier des versions révisées et/ou nouvelles de la Licence publique générale. De telles nouvelles versions seront semblables à la présente version dans l'esprit mais pourront différer dans le détail pour prendre en compte de nouvelles problématiques ou inquiétudes.

Chaque version possède un numéro de version la distinguant. Si le Programme précise le numéro de version de la présente Licence qui s'y applique et "une version ultérieure quelconque", vous avez le choix de suivre les conditions de la présente version ou de toute autre version ultérieure publiée par la Free Software Foundation. Si le Programme ne spécifie aucun numéro de version de la présente Licence, vous pouvez choisir une version quelconque publiée par la Free Software Foundation à quelque moment que ce soit.

10. Si vous souhaitez incorporer des parties du Programme dans d'autres programmes libres dont les conditions de distribution sont différentes, écrivez à l'auteur pour lui en demander l'autorisation. Pour les logiciels dont la Free Software Foundation est titulaire des droits d'auteur, écrivez à la Free Software Foundation ; nous faisons parfois des exceptions dans ce sens. Notre décision sera guidée par le double objectif de préserver le statut libre de tous les dérivés de nos logiciels libres et de promouvoir le partage et la réutilisation des logiciels en général.

Absence de garantie

11. Comme la licence du programme est concédée à titre gratuit, aucune garantie ne s'applique au programme, dans les limites autorisées par la loi applicable. Sauf mention contraire écrite, les titulaires du droit d'auteur et/ou les autres parties fournissent le programme "en l'état", sans aucune garantie de quelque nature que ce soit, expresse ou implicite, y compris, mais sans y être limité, les garanties implicites de commerciabilité et de la conformité à une utilisation particulière. Vous assumez la totalité des risques liés à la qualité et aux performances du programme. Si le programme se révélait défectueux, le coût de l'entretien, des réparations ou des corrections nécessaires vous incombent intégralement.

12. En aucun cas, sauf lorsque la loi applicable ou une convention écrite l'exige, un titulaire de droit d'auteur quel qu'il soit, ou toute partie qui pourrait modifier et/ou redistribuer le programme comme permis ci-dessus, ne pourrait être tenu pour responsable à votre égard des dommages, incluant les dommages génériques, spécifiques, secondaires ou consécutifs, résultant de l'utilisation ou de l'incapacité d'utiliser le programme (y compris, mais sans y être limité, la perte de données, ou le fait que des données soient rendues imprécises, ou les pertes éprouvées par vous ou par des tiers, ou le fait que le programme échoue à interopérer avec un autre programme quel qu'il soit) même si ledit titulaire du droit d'auteur ou la partie concernée a été averti de l'éventualité de tels dommages. Fin des conditions.



Réponses

Cette annexe vous donne les réponses aux quiz et exercices situés à la fin des chapitres. Nous n’avons donné qu’une des solutions possibles pour chacun des problèmes. Dans certains cas, nous avons ajouté des informations supplémentaires pour vous aider à résoudre l’exercice.

Réponses aux questions du Chapitre 1

Quiz

1. Le langage C est puissant, portable, et très répandu.
2. Le compilateur permet de transformer du code source C en langage machine que votre ordinateur pourra comprendre.
3. L'édition, la compilation, la liaison, et le test.
4. La réponse à cette question dépend de votre compilateur. Consultez votre documentation.
5. La réponse à cette question dépend aussi de votre compilateur. Consultez votre documentation.
6. Par convention, l'extension d'un fichier source C est .c.

   Note : C++ utilise l'extension .cpp. Vous pouvez créer et compiler vos programmes C avec cette extension, mais .c est plus approprié.
7. filename.txt pourrait être compilé. Toutefois, l'extension .c serait plus appropriée.
8. Il faut transformer le code source pour corriger les problèmes. Il faudra ensuite recompiler et exécuter la liaison. Relancez votre programme pour vérifier les résultats donnés.
9. Le langage machine est constitué d'instructions numériques, ou binaires, que l'ordinateur peut comprendre. C'est la raison pour laquelle le compilateur doit transformer le code source C en code machine, appelé aussi code objet.
10. L'éditeur de liens associe le code objet de votre programme avec le code objet appartenant à la bibliothèque de fonctions pour créer le fichier exécutable.

Exercices

1. Le fichier objet contient de nombreux caractères bizarres. Au milieu de ces caractères, vous pouvez apercevoir des morceaux de votre fichier source.
2. Le programme calcule l'aire d'un cercle. L'utilisateur doit donner le rayon pour voir apparaître la valeur de l'aire.
3. Ce programme imprime un bloc de 10×10 caractères X. Le Chapitre 6 traite d'un programme similaire.

4. Ce programme génère une erreur de compilation. Vous avez reçu un message semblable à celui-ci :

   ```
   Error: ch1ex4.c: Declaration terminated incorrectly
   ```

   L’erreur provient du point-virgule à la fin de la ligne 3. Si vous l’effacez, la compilation et la liaison s’exécuteront correctement.

5. La compilation de ce programme est bonne, mais il génère une erreur de liaison. Vous avez reçu un message qui ressemble à celui-ci :

   ```
   Error: Undefined symbol _do_it in module...
   ```

   L’éditeur de liens n’a pu trouver une fonction appelée `do it`. Pour corriger ce programme, transformez-la en `printf`.

6. Le programme imprime maintenant un bloc de 10×10 visages souriants.

Réponses aux questions du Chapitre 2

Quiz

1. Un bloc.
2. La fonction `main()`.
3. Tout ce qui apparaît dans un programme entre `/*` et `*/` est un commentaire qui sera ignoré par le compilateur. Ces commentaires permettent de documenter la structure du programme.
4. Une fonction est représentée par son nom et correspond à du code programme qui effectue une certaine tâche. En introduisant ce nom dans un programme, vous exécutez le code associé à cette fonction.
5. Une fonction utilisateur est créée par le programmeur. Une fonction de bibliothèque est fournie avec le compilateur C.
6. Un appel `#include` demande au compilateur d’ajouter le code du fichier désigné au code source, au moment de la compilation.
7. Les commentaires ne doivent pas être imbriqués si l’on veut obtenir un code portable. Même si certains compilateurs permettent de le faire, d’autres ne les acceptent pas.

8. Oui, les commentaires peuvent s'étendre sur plusieurs lignes. Ils commencent avec `/*` et se prolongent jusqu'au `*/` suivant.
9. Un fichier inclus est aussi appelé fichier en-tête.
10. Un fichier inclus est un fichier indépendant contenant les informations nécessaires au compilateur pour utiliser diverses fonctions.

Exercices

1. Rappelez-vous, `main()` est le seul composant obligatoire d'un programme C. Le programme suivant est le plus petit programme possible, mais il ne fait rien :

```
int main()
{
}
```

Vous auriez pu aussi écrire :

```
int main(){ }
```

2. a) Les instructions sont en lignes 8, 9, 10, 12, 20 et 21.
   b) L'unique définition de variable se trouve en ligne 18.
   c) La déclaration de fonction (pour `display line`) se situe ligne 4.
   d) La définition de la fonction `display line` occupe les lignes 16 à 22.
   e) Les lignes 1, 15, et 23 sont des lignes de commentaires.
3. Voici quelques exemples de commentaires:

```
/* ceci est un commentaire */
/*???*/
/*
ceci est un troisième
   commentaire */
```

4. Ce programme imprime l'alphabet en lettres majuscules. Vous comprendrez mieux ce programme lorsque vous atteindrez le Chapitre 10.

   Voici le résultat de l'exécution de ce programme :

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

5. Ce programme compte puis affiche le nombre de caractères et d'espaces que vous saisissez au clavier.

Réponses aux questions du Chapitre 3

Quiz

1. Une variable entière doit contenir un nombre entier (sans chiffres après la virgule) alors que l'on peut stocker dans une variable à virgule flottante n'importe quel nombre réel.
2. Une variable de type `double` a un rang plus élevé qu'une variable de type `float`. Cela signifie que l'on peut y stocker des nombres plus grands et des nombres plus petits. Une variable `double` est aussi plus précise qu'une variable `float`.
3. a) La taille d'un caractère est d'un octet.

   b) La taille d'une variable `short` est inférieure ou égale à celle d'une variable `int`.

   c) La taille d'une variable `int` est inférieure ou égale à celle d'une variable `long`.

   d) La taille d'une variable non signée est égale à la taille d'une variable `int`.

   e) La taille d'une variable `float` est inférieure ou égale à la taille d'une variable `double`.
4. Les noms des constantes symboliques rendent votre source plus facile à lire. Le deuxième avantage est qu'il est très simple de changer leur valeur.
5. Utilisez une des méthodes suivantes :

```
#define MAXIMUM 100
const int MAXIMUM = 100
```

6. Les lettres, les chiffres et les caractères ( ).
7. Les noms de variables et de constantes devraient décrire la donnée qui y sera stockée. Vous pouvez utiliser les lettres minuscules pour les variables, et les majuscules pour les constantes.
8. Les constantes symboliques sont des symboles qui représentent des constantes littérales.
9. Si c'est une variable de type `unsigned short`, c'est-à-dire d'une longueur de 2 octets, la valeur minimale que l'on peut y stocker est 0. Si c'est une variable de type `signed`, sa valeur minimum est –32768.

Exercices

1. Voici les réponses :

   a) L'âge d'une personne étant un nombre entier qui ne peut être négatif, nous suggérons une variable de type `unsigned char` (entre 0 et 255).

b) `unsigned short`.

c) `float`.

d) Si vos prévisions ne sont pas très élevées, une simple variable `unsigned short` fera l'affaire. Si vous pensez pouvoir dépasser les 65535 euros, utilisez plutôt une variable de type `long` et pensez à parler de nous à votre employeur !

e) `float` (n'oubliez pas les centimes après la virgule).

f) La note la plus haute étant supposée être toujours égale à 100, c'est une constante. Utilisez une instruction `const int` ou `#define`.

g) `float` (si vous ne choisissez que des nombres entiers, utilisez `int` ou `long`).

h) Une variable `signed`. Le type sera `int`, `long`, ou `float`.

i) double.

2. Voici la réponse aux exercices 2 et 3 :

   Le nom de la variable représente la valeur qui y sera stockée. Une déclaration est une instruction qui crée la variable et qui peut l'initialiser. Vous ne pouvez pas utiliser les mots clé du langage C.

   a) `unsigned short age;`.

   b) `unsigned short poids;`.

   c) `float rayon = 3;`.

   d) `long salaire annuel;`.

   e) `float cout = 29.95;`.

   f) `const int note max = 100;` ou `#define NOTE MAX 100`.

   g) `float temperature;`.

   h) `long gain =  30000;`.

   i) `double distance;`.

3. Voir les réponses ci-avant.

4. Les noms corrects sont : b, c, e, g, h, i, et j.

   Le nom donné en j est correct, mais utiliser un nom aussi long est très lourd et beaucoup de compilateurs ignoreront les derniers caractères.

   Voici les noms incorrects :

   a) Un nom de variable ne peut pas commencer par un chiffre.

   d) Le signe (#) est interdit.

   f) Le tiret ( ) est interdit.

Réponses aux questions du Chapitre 4

Quiz

1. C'est une instruction d'affectation qui demande à l'ordinateur d'ajouter 5 et 8, puis d'attribuer le résultat à la variable `x`.
2. Une expression est quelque chose qui a une valeur numérique.
3. La hiérarchie de ces opérateurs.
4. Après la première instruction, la valeur de `a` est 10 et celle de `x` est 11. Après la deuxième instruction, les deux variables `a` et `x` ont la valeur 11.
5. `1`
6. `19`
7. `(5+3) * 8 / (2+2)`
8. `0`
9. Vous trouverez la hiérarchie des opérateurs à la fin de ce chapitre.
   a) <.
   b) *.
   c) `!=` et == ayant la même priorité, ils seront traités de gauche à droite.
   d) >= et > ont la même priorité.
10. Les opérateurs d'affectation composé permettent d'effectuer une opération mathématique binaire et une opération d'affectation simultanément. Les opérateurs de ce type présentés dans ce chapitre sont : (+=), (–=), (/=), (*=) et(%=).

Exercices

1. Ce listing est correct d'un point de vue syntaxique, mais il est très difficile à lire. Son objectif est de démontrer que les blancs n'ont aucune influence sur l'exécution d'un programme. Utilisez largement les lignes blanches et les tabulations pour rendre le code de vos programmes compréhensible.
2. Voici le code de l'exercice 1 mieux structuré :

```
#include <stdio.h>
#include <stdlib.h>

int x, y;
int main()
{
```

```
   printf("\nEntrez deux nombres ");
   scanf("%d %d", &x, &y);
   printf("\n\n%d est plus grand\n", (x>y)?x:y);
   exit(EXIT_SUCCESS);
}
```

3. Voici les lignes de Listing 4.1 qu'il faut modifier :

```
16:    printf("\n%d    %d", a++, ++b);
17:    printf("\n%d    %d", a++, ++b);
18:    printf("\n%d    %d", a++, ++b);
19:    printf("\n%d    %d", a++, ++b);
20:    printf("\n%d    %d", a++, ++b);
```

4. Les instructions suivantes représentent une des nombreuses solutions possibles. Elles testent si `x` est plus grand ou égal à 1, et si `x` est inférieur ou égal à 20. Si ces deux conditions sont réunies, la valeur de `x` est attribuée à `y`. Sinon, la valeur de `x` n'est pas attribuée à `y`.

```
if ((x>=1)&&(x<=20))
   y=x;
```

5. L'instruction devient :

```
y = ((x>=1) && (x<=20)) ? x : y;
```

6. Le code est le suivant :

```
if (x<1 && x>10)
   instruction;
```

7. Voici les réponses :
   a) 7.
   b) 0.
   c) 9.
   d) 1 (vrai).
   e) 5.
8. Voici les réponses :
   a) Vrai.
   b) Faux.
   c) Vrai. Notez qu'il n'y a qu'un signe égale, l'instruction `if` est utilisée comme une instruction d'affectation.

9. Voici une solution :

```
if (age<21)
   printf("Vous n'êtes pas un adulte");
else if(age>=65)
   printf("Vous êtes une personne âgée");
   else
     printf("Vous êtes un adulte");
```

10. Ce programme comporte quatre erreurs. La première se trouve en ligne 4, cette instruction aurait dû se terminer par un point-virgule. La deuxième est le point-virgule à la fin de l'instruction `if` en ligne 7. La troisième est très courante : l'opérateur (=) a été utilisé à la place de (==) dans l'instruction `if`. La dernière erreur est le mot sinon en ligne 9.

    Voici le code corrigé :

```
/* programme bogué */
#include <stdio.h>
#include <stdlib.h>
int x = 1;
int main()
{
   if(x == 1)
   printf("x égal 1");
   else
   printf("x n'est pas égal à 1");

   exit(EXIT_SUCCESS);
}
```

Réponses aux questions du Chapitre 5

Quiz

1. Répondez oui si vous voulez devenir un bon programmeur de C.
2. La programmation structurée prend en compte un problème de programmation complexe, et le divise en plusieurs tâches plus simples. Ces tâches seront plus faciles à programmer une par une.
4. La première ligne d'une définition de fonction est l'en-tête. Il contient le nom de la fonction, le type de la valeur renvoyée, et la liste de paramètres.
5. Une fonction peut renvoyer une valeur ou pas de valeur du tout. Cette valeur appartient à n'importe quel type de variable du langage C.
6. Le type de valeur renvoyée pour une fonction qui ne renvoie rien est `void`.

7. La définition de fonction représente la fonction en elle-même, et comprend l'en-tête et les instructions. La définition détermine les opérations qui seront exécutées à l'appel de la fonction. Le prototype est une simple ligne de code, identique à l'en-tête de la fonction, qui se termine par un point-virgule. Ce prototype donne au compilateur le nom de la fonction, le type de valeur renvoyée par cette fonction, et la liste de paramètres.
8. Une variable locale est déclarée dans une fonction.
9. Les variables locales sont indépendantes des autres variables du programme.
10. Peu importe. Cependant, pour la localiser plus facilement, placez-la soit au début de votre programme comme nous le faisons tout au long de ce livre, soit tout à la fin.

Exercices

1. float fait_le(char a, char b, char c)
2. void affiche_un_nombre(int un_nombre)
3. Les valeurs renvoyées sont du type :
   a) `int`.
   b) `long`.
4. Ce code contient deux erreurs. La première est la déclaration de la fonction avec `void` alors qu'elle renvoie une valeur. L'instruction `return` doit être supprimée. La deuxième erreur est à la ligne 6. L'appel de la fonction `print msg()` contient un paramètre (une chaîne). Le prototype indiquait que la liste de paramètres de cette fonction était de type `void`. Cela signifie qu'il n'y a pas de paramètre à transmettre. Voici les instructions corrigées :

```
#include <stdio.h>
#include <stdlib.h>
void print_msg (void);
int main()
{
   print_msg();
   exit(EXIT_SUCCESS);
}
void print_msg(void)
{
puts("Cela est un message à afficher");
}
```

5. Il ne doit pas y avoir de point-virgule à la fin de l'en-tête.

6. La fonction `larger of` est la seule qui ait besoin d'être modifiée :

```
int larger_of(int a, int b)
{
  int save;


  if (a>b)
  save = a;
  else
  save = b;

  return save;
}
```

7. Dans cette fonction, nous supposons que les deux valeurs sont entières et que la valeur renvoyée est entière :

```
int produit(int x, int y)
{
   return (x * y);
}
```

8. Le listing qui suit vérifie si la seconde valeur transmise est bien différente de zéro, une division par zéro provoque une erreur. Vous devez toujours contrôler les valeurs transmises :

```
int divise(int a, int b)
{
   int reponse = 0;
   if(b == 0)
    reponse = 0;
   else
    reponse = a/b;
   return reponse;
}
```

9. La fonction utilisée pour cette solution est `main()`, mais cela aurait pu être une autre fonction. Les lignes 10, 11 et 12 contiennent les appels des deux fonctions. Les lignes 14 à 17 affichent les valeurs. Pour exécuter ce programme, vous devez ajouter le code des exercices 7 et 8 après la ligne 20 :

```
 1:  #include <stdio.h>
 2:  #include <stdlib.h>
 3:
 4:  int main()
 5:  {
 6:     int nombre1 = 10;
 7:     nombre2 = 5;
```

```
    int x, y, z;

    x = produit(nombre1, nombre2);
    y = divise(nombre1, nombre2);
    z = divise(nombre1, 0);

   printf("\nnombre1 est %d et nombre2 est %d, nombre1, nombre2);
   printf("\nnombre1 * nombre2 a la valeur %d", x);
   printf("\nnombre1 / nombre2 a la valeur %d", y);
   printf("\nnombre1 / 0 a la valeur %d", z);

   exit(EXIT_SUCCESS);
}
```

10. Voici une solution :

```
/* Calcul de la moyenne des cinq nombres entrés par l'utilisateur */
#include <stdio.h>
#include <stdlib.h>
float v, w, x, y, z, reponse;
float moyenne(float a, float b, float c, float d, float e);
int main()
{
   puts("Entrez cinq nombres :");
   scanf("%f%f%f%f%f", &v, &w, &x, &y, &z);
   reponse = moyenne(v, w, x, y, z);
   printf(("La moyenne est %f.\n", reponse);

   exit(EXIT_SUCCESS);
}
float moyenne(float a, float b, float c, float d, float e)
{
   return((a+b+c+d+e)/5);
}
```

11. Cette réponse contient des variables de type `int`. Leurs valeurs devront être inférieures à 9 :

```
/* Ce programme utilise une fonction récursive */
#include <stdio.h>
#include <stdlib.h>
int trois_puissance(int puissance);
int main()
{
   int a = 4;
   int b = 9;
   printf("\n3 à la puissance %d vaut %d", a, trois_puissance(a);
   printf("\n3 à la puissance %d vaut %d\n", b, trois_puissance(b);

   exit(EXIT_SUCCESS);
}
```

```
int trois_puissance(int puissance)
{
   if (puissance < 1);
   return(1);
   else
   return(3 * trois_puissance(puissance-1));
}
```

Réponses aux questions du Chapitre 6

Quiz

1. La première valeur de l'index d'un tableau en langage C est 0.
2. Les expressions d'initialisation et d'incrément font partie de la commande `for`.
3. L'instruction `while` de la boucle `do while` se situe à la fin de la commande et s'exécute toujours au moins au fois.
4. Une instruction `while` peut effectivement accomplir les mêmes tâches que `for`. Vous avez toutefois deux opérations supplémentaires à exécuter. Il faut initialiser certaines variables avant de lancer la commande `while`, et incrémenter les variables nécessaires dans la boucle.
5. Le recouvrement des boucles est interdit. La boucle imbriquée doit se trouver entièrement dans la boucle extérieure.
6. Oui. Toutes les commandes du langage C peuvent être imbriquées les unes dans les autres.
7. Les quatre parties d'une instruction `for` sont l'initialisation, la condition, l'incrémentation et les instructions.
8. Une instruction `while` est constituée d'une condition suivie d'instructions.
9. Une instruction `do...while` est constituée d'une condition suivie d'instructions.

Exercices

1. long tableau[50];
2. tableau[49] = 123.456;
3. La valeur de `x` est 100.
4. La valeur de `ctr` sera 11.
5. La boucle imbriquée affiche 5 caractères X. La boucle extérieure affiche 10 fois la boucle imbriquée. Cela donne un total de 50 caractères X.

6. Réponse :

```
int x;
for(x = 1; x <= 100; x +=3);
```

7. Réponse :

```
int x = 1;
while(x <= 100)
   x += 3;
```

8. Réponse :

```
int ctr = 1;
do
{
   ctr +=3;
} while (ctr < 100);
```

9. Ce programme ne se terminera jamais. `record` est initialisé à zéro. La boucle `while` s'assure ensuite que `record` est inférieur à 100. 0 étant inférieur à 100, la boucle s'exécute et affiche les deux messages. La boucle contrôle de nouveau la variable `record` dont la valeur ne changera jamais. La boucle va donc continuer à s'exécuter. La ligne suivante aurait dû être ajoutée après la deuxième instruction `printf()` :

```
record++;
```

10. L'erreur dans ce code est le point-virgule à la fin de l'instruction `for`.

Réponses aux questions du Chapitre 7

Quiz

1. Les fonctions `puts()` et `printf()` présentent deux différences :
   - `printf()` peut afficher des variables.
   - `puts()` ajoute un caractère de retour à la ligne à la fin de la chaîne à émettre.
2. Le fichier en-tête stdio.h.
3. a) `\\` affiche un antislash (`\`).
   b) `\b` envoie un retour arrière.
   c) `\n` envoie un retour à la ligne.
   d) `\t` envoie une tabulation.
   e) `\a` fait bipper l'ordinateur.

4. a) `%s` pour une chaîne de caractères.
   b) `%d` pour un entier décimal signé.
   c) `%f` pour un nombre décimal avec virgule flottante.
5. a) `b` affichera le caractère b.
   b) `\b` effectue un retour arrière du curseur.
   c) `\` avec le caractère qui suit représente un ordre de contrôle.
   d) `\\` affiche un antislash (`\`).

Exercices

1. Contrairement à la fonction `printf()`, la fonction `puts()` ajoute automatiquement le retour à la ligne :

```
printf("\n");
puts("");
```

2. Réponse :

```
char c1, c2;
int d1;
scanf("%c %ud %c", &c1, &d1, &c2);
```

3. Une réponse :

```
#include <stdio.h>
#include <stdlib.h>
int x;
int main()
{
   puts("Entrez une valeur entière :");
   scanf("%d", &x);
   printf("\nLa valeur entrée est %d\n", x);

   exit(EXIT_SUCCESS);
}
```

4.

```
#include <stdio.h>
#include <stdlib.h>
int x;
int main()
{
   puts("Entrez une valeur entière paire :");
   scanf("%d", &x);
   while(x % 2 != 0)
   {
```

```
    printf("\n%d n'est pas pair, entrez un nombre pair SVP :", x);
    scanf("%d", &x);
  }
  printf("\nLa valeur entrée est %d\n", x);

  exit(EXIT_SUCCESS);
}
```

5. Réponse :

```
#include <stdio.h>
#include <stdlib.h>
int tableau[6], x, nombre;
int main()
{
  /* Boucle 6 fois ou jusqu'à lecture de l'élément 99 */
  for(x=0; x<6 && nombre !=99; x++)
  {
    puts("Entrez une valeur entière paire, ou 99 pour sortir :");
    scanf("%d", &nombre);
    while(nombre % 2 == 1 && nombre != 99)
    {
    printf("\n%d n'est pas pair, entrez un nombre pair SVP :", nombre);
    scanf("%d", &nombre);
    }
    tableau[x] = nombre;
  }
/* affichage des résultats */
  for(x=0; x<6 && tableau[x] != 99; x++)
  {
    printf("\nLa valeur entrée est %d", tableau[x]);
  }

  exit(EXIT_SUCCESS);
}
```

6. Il faut modifier la dernière fonction `printf()` :

```
printf("%d\t", tableau[x]);
```

7. Vous ne pouvez pas inclure de guillemets dans un texte entre guillemets. Pour afficher des guillemets, vous devez utiliser le caractère (\). De plus, vous devez ajouter un slash (/) à la fin de la première ligne pour que le texte puisse continuer sur la seconde :

```
printf("Jacques a dit, \ "Levez le bras \
droit ! \"");
```

8. Ce code contient trois erreurs. La première est l'absence de guillemets dans la fonction `printf()`. La deuxième est l'absence de l'opérateur d'adresse avec la variable

réponse de `scanf()`. La dernière erreur est toujours dans cette fonction. Il faut `%d` et non `%f` pour la variable réponse. Voici le code corrigé :

```
int lire_1_ou_2(void)
{
   int reponse = 0;
   while (reponse < 1 || reponse > 2)
   {
     printf("Entrez 1 pour oui, 2 pour non");
     scanf("%d", &reponse);
   }
   return reponse;
}
```

9. Voici ce que vous devrez ajouter à Listing 7.1 :

```
void affiche(void)
{
  printf("\nExemple d'affichage");
  printf("\n\nOrdre\tSignification");
  printf("\n======\t==============");
  printf("\n\\a\t\tsonnerie ");
  printf("\n\\b\t\tretour arrière");
  printf("\n\\n\t\tretour à la ligne");
  printf("\n\\t\t\ttabulation horizontale");
  printf("\n\\\\\t\tantislash");
  printf("\n\\?\t\tpoint d'interrogation");
  printf("\n\\'\t\tguillemet simple");
  printf("\n\\"\t\tguillemet double");
  printf("\n...\t\t...");
}
```

10. Réponse :

```
/* Lecture de deux nombres avec virgule flottante */
/* puis affichage de leur produit */
#include <stdio.h>
#include <stdlib.h>
float x, y,;
int main()
{
   puts("Entrez deux valeurs :");
   scanf("%f %f", &x, &y);
   printf("\nLe produit est %f\n", x*y);
   exit(EXIT_SUCCESS);
}
```

11. Réponse :

```
/* Lecture de 10 entiers et affichage de leur somme */
#include <stdio.h>
#include <stdlib.h>
int count, temp;
```

```
long total = 0;
int main()
{
   for (count = 1; count <= 10; count++)
    {
    printf("Entrez un entier # %d :", count);
    scanf("%d", &temp);
    total +=temp;
    }
   printf("\n\nLe total est %d\n", total);

   exit(EXIT_SUCCESS);
}
```

12. Réponse :

```
#define MAX 100

int tableau[MAX];
int count = -1, maximum, minimum, nombre_saisi, temp;

int main()
{
    puts("Entrez une valeur entière par ligne, 0 pour finir : ");

    /* lecture des valeurs */

   do
   {
    scanf("%d", &temp);
    tableau[++count] = temp;
   }while (count < (MAX-1) && temp != 0);

   nombre_saisi = count;

   /* Recherche du plus petit et du plus grand */

   maximum = -32000;
   minimum = 32000;

   for(count=0; count<=nombre_saisi && tableau[count] != 0; count++)
   {
    if(tableau[count] > maximum)
     maximum = tableau[count];

    if(tableau[count] < minimum)
     minimum = tableau[count];
   }

   printf("\nLa valeur maxi est %d", maximum);
   printf("\nLa valeur mini est %d\n", minimum);
   exit(EXIT_SUCCESS);

}
```

Réponses aux questions du Chapitre 8

Quiz

1. N'importe quel type de donnée, mais un seul à la fois. Un tableau ne peut contenir des données de types différents.
2. `0`.
3. `n    1`.
4. Le programme sera compilé et exécuté, mais les résultats seront imprévisibles.
5. Dans l'instruction de déclaration, le nom de tableau sera suivi d'une paire de crochets pour chaque dimension. Chacune de ces paires contiendra le nombre d'éléments de la dimension correspondante.
6. 240. Ce résultat est obtenu en multipliant $2 \times 3 \times 5 \times 8$.
7. `tableau[0] [0] [1] [1]`.

Exercices

1. `int un[1000], deux[1000] trois[1000];`.
2. `int tableau[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};`.
3. Cet exercice peut être résolu de multiples façons. La première solution consiste à initialiser le tableau avec la déclaration :

```
int huitethuit[88] = {88, 88, 88, 88, 88,...,88};
```

Cette solution impose l'écriture de 88 nombres `88` entre les accolades. La méthode suivante est meilleure :

```
int huitethuit[88];
int x;
for (x=0; x<88; x++)
   huitethuit[x] = 88;
```

4. Réponse :

```
int stuff[12][10];
int sub1, sub2;
for(sub1=0; sub1<12; sub1++)
   for(sub2=0; sub2<10; sub2++)
     stuff[sub1][sub2]=0;
```

5. Attention, cette erreur est très facile à commettre. Le tableau de 10 ∞ 3 éléments a été initialisé comme un tableau de 3 ∞ 10 éléments. Il y a deux méthodes pour corriger ce code. La première consiste à inverser `x` et `y` dans la ligne d'initialisation :

```
int x, y;
int tableau[10][3];
int main()
{
for (x=0; x<3; x++)
   for (y=0; y<10; y++)
     tableau[y][x]=0;          /* modifié */

   exit(EXIT_SUCCESS);
}
```

La seconde méthode (celle qui est recommandée) est d'inverser les valeurs de la boucle `for` :

```
int x, y;
int tableau[10][3];
int main()
{
for (x=0; x<10; x++)
   for (y=0; y<3; y++)
     tableau[y][x]=0;
   exit(EXIT_SUCCESS);
}
```

6. Ce programme initialise les éléments du tableau ayant des valeurs d'index de 1 à 10. Or, un tableau de 10 éléments a des valeurs d'index comprises entre 0 et 9. L'élément `tableau[10]` n'existe pas. L'instruction `for` doit avoir une des deux formes suivantes :

```
for(x=1; x<=9; x++)    /* initialise 9 éléments sur 10 */
for(x=0; x<=9; x++)
```

7. Voici l'une des nombreuses réponses possibles :

```
/* utilisation d'un tableau à 2 dimensions et de rand() */

#include <stdio.h>
#include <stdlib.h>

/* déclaration du tableau */

int tableau[5][4];
int a, b;

int main()
{
   for (a=0; a<5; a++)
   {
     for (b=0; b<4; b++)
```

```
      {
      tableau[a][b] = rand();
      }
    }
    /* Affichage des éléments du tableau */
    for (a=0; a<5; a++)
    {
     for (b=0; b<4; b++)
     {
      printf("%d\t", tableau[a][b]);
     }
     printf("\n");
    }
    exit(EXIT_SUCCESS);
}
```

8. Voici l'une des nombreuses réponses possibles :

```
/* random.c. Utilisation d'un tableau à une dimension. */

#include <stdio.h>
#include <stdlib.h>
/* Déclaration d'un tableau à 1 dimension avec 1000 éléments */

int random[1000];
int a, b, c;
long total = 0;

int main()
{
/* On remplit le tableau avec des nombres aléatoires. La fonction */
/* de bibliothèque rand() renvoie un nombre aléatoire.*/
/* On utilise une boucle for pour chaque valeur d'index du tableau. */

   for (a = 0; a < 1000; a++)
   {
   random[a] = rand();
   total += random[a];
   }
   printf("\n\nLa moyenne est :  %ld\n", total/1000);
   /* On affiche les éléments du Tableau 10 par 10 */

   for (a = 0; a < 1000; a++)
   {
    printf("\nrandom[%d] = ", a);
    printf("%d", random[a]);
     if (a%10 == 0 && a>0)
     {
     printf("\n Appuyez sur Entrée pour continuer.");
     printf("\nou CTRL-C pour sortir.");
     getchar();
     }
   }
   exit(EXIT_SUCCESS);
}  /* fin de la fonction main() */
```

9. Voici deux solutions. La première initialise le tableau au moment de sa déclaration et la seconde l'initialise dans une boucle `for`.

   Première solution :

```
#include <stdio.h>
#include <stdlib.h>

/* Déclaration d'un tableau à une dimension */
6 :
int elements[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int idx;

int main()
{
    for (idx = 0; idx<10; idx++);
    {
     printf("\nelements[%d] = %d", idx, elements[idx]);
    }
    exit(EXIT_SUCCESS);

}
```

   Seconde solution :

```
#include <stdio.h>
#include <stdlib.h>

/* Déclaration d'un tableau à une dimension */

int elements[10];
int idx;

int main()
{
   for (idx = 0; idx < 10; idx++)
      elements[idx] = idx ;

   for (idx = 0; idx < 10; idx++)
      printf( "\nelements[%d] = %d ", idx, elements[idx] );

   exit(EXIT_SUCCESS);
}
```

10. Les instructions suivantes représentent l'une des solutions :

```
#include <stdio.h>
#include <stdlib.h>

/* Déclaration d'un tableau à une dimension */

int elements[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int nouv_tableau[10];
int idx;
```

```
int main()
{
   for (idx = 0; idx<10; idx++)
   {
    nouv_tableau[idx] = elements[idx] +  10;
   }

   for (idx = 0; idx<10; idx++)
   {
    printf("\nelements[%d] = %d \tnouv_tableau[%d] = %d",
    idx, elements[idx], idx, nouv_tableau[idx]);
   }
   exit(EXIT_SUCCESS);
}
```

Réponses aux questions du Chapitre 9

Quiz

1. L'opérateur d'adresse (`&`).
2. L'opérateur indirect (`*`). Si cet opérateur précède le nom d'un pointeur, il fait référence à la variable pointée.
3. Un pointeur est une variable qui contient l'adresse d'une autre variable.
4. On utilise une indirection quand on accède à la valeur d'une variable avec le pointeur de cette variable.
5. Ils sont stockés dans des emplacements mémoire adjacents, les premiers éléments ayant les adresses les plus basses.
6. `&data[0]`

   ```
   data
   ```
7. La première méthode consiste à transmettre la taille du tableau en paramètre ; la seconde, à placer une valeur particulière dans le dernier élément du tableau.
8. Affectation, indirection, adresse de, incrémentation, différence et comparaison.
9. La différence entre deux pointeurs donne le nombre d'éléments qui les séparent. Dans notre cas, la réponse est 1. La taille réelle des éléments du tableau n'a aucune influence.
10. La réponse est toujours 1.

Exercices

1. `char *ptr char;`.
2. Les instructions suivantes déclarent le pointeur de `cout`, et l'initialise avec l'adresse de la variable :

```
int *p_cout;
p_cout = &cout;
```

3. Accès direct : `cout = 100;`.

   Accès indirect : `*p cout = 100;`.
4. `printf( "La valeur du pointeur %d, est l'adresse de la variable %d", p cout, *p cout);`.
5. `float *variable = &radius;`.
6. Réponses :

```
data[2] = 100;
*(data + 2) = 100;
```

7. Ce code contient aussi la réponse à l'exercice 8 :

```
/* ex9_7.c */
#include <stdio.h>
#include <stdlib.h>

#define MAX1 5
#define MAX2 8

int tableau1[MAX1] = { 1, 2, 3, 4, 5};
int tableau2[MAX2] = { 1, 2, 3, 4, 5, 6, 7, 8};
int total;

int somtabs(int x1[], int len_x1, int x2[], int len_x2);

int main()
{
  total = somtabs(tableau1, MAX1, tableau2, MAX2);
  printf("Le total est %d\n", total);
  exit(EXIT_SUCCESS);

}

int somtabs(int x1[], int len_x1, int x2[], int len_x2)
{
  int total = 0, count = 0;

  for (count = 0; count < len_x1; count++)
   total += x1[count];
```

```

   for (count = 0; count < len_x2; count++)
    total += x2[count];

   return total;
}
```

8. Voir la réponse de l'exercice 7.
9. Voici une réponse :

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

/* prototypes des fonctions */
void addarrays( int [], int []);

int main()
{
   int a[SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
   int b[SIZE] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

   addarrays(a, b);

   exit(EXIT_SUCCESS);
}

void addarrays( int first[], int second[])
{
    int total[SIZE];
    int *ptr_total = &total[0];
    int ctr = 0;

    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       total[ctr] = first[ctr] +  second[ctr];
       printf("%d + %d = %d\n", first[ctr], second[ctr], total[ctr]);
   }
}
```

Réponses aux questions du Chapitre 10

Quiz

1. Les valeurs du code ASCII sont comprises entre 0 et 255. Les valeurs 0 à 127 représentent les caractères standards, les valeurs 128 à 255 sont les caractères étendus.
2. Comme le code ASCII d'un caractère.

3. Une chaîne est une séquence de caractères terminée par le caractère nul.
4. Une séquence de caractères entre des guillemets doubles.
5. Pour stocker le caractère nul de fin.
6. Il interprète une chaîne littérale comme une séquence des codes ASCII correspondants suivie de 0 (code ASCII du caractère nul).
7. a) 97.
   b) 65.
   c) 57.
   d) 32.
   e) 206.
   f) 6.
8. a) I.
   b) Un blanc.
   c) c.
   d) a.
   e) n.
   f) NUL.
   g) B.
9. a) 9 octets.
   b) 9 octets.
   c) 1 octet.
   d) 20 octets.
   e) 20 octets.
10. a) U.
    b) U.
    c) 0 (nul).
    d) Cette valeur se trouve en dehors de la chaîne.
    e) !.
    f) Cette variable représente l'adresse du premier élément de la chaîne.

Exercices

1. `char lettre = '$';`.
2. `char tableau[26] = "les pointeurs sont fous!";`.
3. `char *tableau = "les pointeurs sont fous!";`.
4. Réponse(la fonction `lire clavier()` a été définie au début de l'ouvrage) :

```
char *ptr;
ptr = malloc(81);
lire_clavier(ptr, 81 * sizeof(*ptr));
```

5. Le programme complet suivant représente l'une des solutions :

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

/* prototypes des fonctions */
void copyarrays( int [], int []);

int main()
{
    int ctr=0;
    int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[SIZE];

    /* valeurs avant la copie */
    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       printf( "a[%d] = %d, b[%d] = %d\n",
               ctr, a[ctr], ctr, b[ctr]);
    }

    copyarrays(a, b);

    /* valeurs après la copie */
    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       printf( "a[%d] = %d, b[%d] = %d\n",
               ctr, a[ctr], ctr, b[ctr]);
    }

    exit(EXIT_SUCCESS);
}

void copyarrays( int orig[], int newone[])
{
    int ctr = 0;

    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
        newone[ctr] = orig[ctr];
    }
}
```

6. Voici une réponse :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* prototypes des fonctions */
char * compare_strings( char *, char *);

int main()
{
    char *a = "Hello";
    char *b = "World!";
    char *longer;

    longer = compare_strings(a, b);

    printf( "Voici la chaîne la plus longue: %s\n", longer );

    exit(EXIT_SUCCESS);
}

char * compare_strings( char * first, char * second)
{
    int x, y;

    x = strlen(first);
    y = strlen(second);

    if( x > y)
       return(first);
    else
       return(second);
}
```

7. Cet exercice est libre.

8. `une chaine` est déclarée comme un tableau de dix caractères. Mais la chaîne d'initialisation est plus longue !

9. Si cette ligne de code est destinée à initialiser une chaîne, c'est une erreur. Il faut utiliser `char *quote` ou `char quote[100]`.

10. Non.

11. Oui. Il est impossible d'affecter un tableau dans un autre tableau.

Réponses aux questions du Chapitre 11

Quiz

1. Les données d'un tableau sont toutes du même type. Une structure peut contenir des données de types différents.
2. L'opérateur (`.`) est utilisé pour accéder aux membres d'une structure.
3. `struct`.
4. Le nom du type de la structure ne représente qu'un modèle de structure, ce n'est pas une variable. Un nom de structure est une variable structure pour laquelle on a réservé de la mémoire.
5. Ces instructions définissent un modèle et déclarent une structure appelée monadresse. Elle est ensuite initialisée : le membre monadresse.nom prend la valeur "Bradley Jones", monadresse.adr1 prend la valeur "RTSoftware", monadresse.adr2 est initialisé à "P.O. Box 1213", monadresse.ville prend la valeur "Carmel", etc.
6. L'instruction qui suit modifie `ptr` pour le faire pointer sur le deuxième élément du tableau : `ptr++;`

Exercices

1. Réponse :

```
struct time {
   int hours;
   int minutes;
   int seconds;
};
```

2. Réponse :

```
struct data {
   int value1;
   float value2;
   float value3;
} info;
```

3. `info.value1` = 100;
4. Réponse :

```
struct data *ptr;
ptr = &info;
```

5. Réponse :

```
ptr -> value2 = 5.5;
(*ptr).value2 = 5.5;
```

6. Réponse :

```
struct data {
   char nom[21];
   struct data *ptr;
};
```

7. Réponse :

```
typedef struct {
   char adresse1[31];
   char adresse2[31];
   char ville[11];
   char etat[3];
   char zip[11];
} RECORD;
```

8. L'instruction suivante utilise les valeurs de la question 5 du quiz pour l'initialisation :

```
RECORD monadresse = {"RTSoftware",
   "P.O. Box 1213",
   "Carmel", "IN", "46032-1213"};
```

9. Ces instructions contiennent deux erreurs. La première est que le modèle de structure n'a pas de nom. La seconde est la façon dont `signe` a été initialisé : il manque les accolades. Voici le code corrigé :

```
struct zodiac {
   char signe_zodiaque[21];
   int mois;
} signe = { "lion", 8};
```

10. Cette déclaration ne comporte qu'une erreur. On ne peut utiliser qu'une variable à la fois dans une union. L'initialisation n'est autorisée que pour le premier membre de l'union. Voici le code corrigé :

```
/* création d'une union */
union data{
   char un_mot[4];
   long nombre;
}variable_generic = {"WOW"};
```

Réponses aux questions du Chapitre 12

Quiz

1. La portée de la variable représente la partie du programme qui a accès à cette variable.
2. Une variable qui appartient à une classe de stockage locale n'est visible que dans la fonction où elle est définie. Une variable avec une classe de stockage externe est visible dans tout le programme.
3. Si une variable est définie dans une fonction, elle est locale. Si cette définition n'est pas dans une fonction, la variable est externe.
4. Automatique (par défaut) et statique. Une variable automatique sera créée à chaque appel de la fonction, et détruite dès la fin de son exécution. Les variables locales statiques persistent et gardent leur valeur entre deux appels de la fonction.
5. Une variable automatique est initialisée à chaque appel de la fonction. Une variable statique n'est initialisée qu'au premier appel de la fonction.
6. Faux. En déclarant une variable de type `register`, vous émettez une demande. Il n'y a aucune garantie pour que le système accède à cette demande.
7. Une variable globale est automatiquement initialisée à 0. Il est tout de même préférable d'initialiser toutes vos variables de façon explicite.
8. Une variable locale n'est pas initialisée automatiquement. Elle peut contenir n'importe quelle valeur.
9. La variable `count` étant maintenant locale pour le bloc, la fonction `printf()` n'y a plus accès . Le compilateur générera un message d'erreur.
10. Elle doit être déclarée avec le mot clé `static`.
11. Le mot clé `extern` est un attribut de classe de stockage. Il indique que la variable a été déclarée quelque part ailleurs dans le programme.
12. Le mot clé `static` est un attribut de classe de stockage. Il demande au compilateur de conserver la valeur de la variable pendant toute la durée de l'exécution du programme. Dans une fonction, la variable conservera sa valeur entre deux appels de la fonction.

Exercices

1. Réponse :

```
/* Démonstration de la portée d'une variable. */
#include <stdlib.h>
#include <stdio.h>
```

```
void print_value(int x);
int main()
{
    int x = 999;
   printf("%d", x);
   print_value(x);
   exit(EXIT_SUCCESS);
}
void print_value(int x)
{
   printf("%d", x);
}
```

2. La variable `var` est une variable globale. Vous n'avez donc pas besoin de la transmettre dans la liste de paramètres.

```
/* Utilisation d'une variable globale */
#include <stdio.h>
#include <stdlib.h>
int var = 99;
void print_value(void);
int main()
{
   print_value();
   exit(EXIT_SUCCESS);
}
void print_value(void)
{
   printf("la valeur est %d\n", var);
}
```

3. Oui, vous devez transmettre la variable `var` si vous voulez l'utiliser dans une autre fonction.

```
/* Utilisation d'une variable globale */
#include <stdio.h>
#include <stdlib.h>
void print_value(int var);
int main()
{
int var = 99;
   print_value(var);
   exit(EXIT_SUCCESS);
}
void print_value(int var)
{
   printf("la valeur est %d\n", var);
}
```

4. Oui, un programme peut avoir une variable globale et une variable locale de même nom. C'est la variable locale qui prendra le pas sur l'autre.

```
/* Utilisation de variables globale et locale*/
#include <stdio.h>
#include <stdlib.h>
int var = 99;
void print_value(void);
int main()
{
   int var = 77;
   printf("Affichage de la valeur dans une fonction ou var est locale");
   printf("et globale.\nLa valeur de var est %d", var);
   print_value();
   exit(EXIT_SUCCESS);
}
void print_value(void)
{
   printf("Affichage de la valeur dans une fonction ou var est globale");
   printf("la valeur de var est %d\n", var);
}
```

5. Il n'y a qu'une erreur avec la fonction `exemple de fonction()`. Les variables peuvent être déclarées au début d'un bloc. Les déclarations de `ctr1` et `star` sont correctes. La variable `ctr2` n'a pas été déclarée au début du bloc, elle doit l'être. Le programme qui suit utilise la fonction corrigée.

   Note : si votre compilateur est un compilateur C++, il acceptera le programme avec son erreur. C++ applique en effet des règles différentes concernant l'emplacement de la déclaration des variables. Vous devez cependant suivre les règles du langage C même si votre compilateur permet de travailler différemment.

```
#include <stdio.h>
#include <stdlib.h>

void exemple_de_fonction( );
int main()
{
    exemple_de_fonction();
    exit(EXIT_SUCCESS);
}

void exemple_de_fonction(void)
{
   int ctr1;
   for (ctr1 = 0; ctr1 < 25; ctr1++)
     printf("*");
   puts ("Ceci est un exemple de fonction\n");
   {
     char star = '*';
     int ctr2;   /* correction */
     puts("il y a un problème\n");
```

```
        for (int ctr2 = 0; ctr2 < 25; ctr2++)
        {
        printf("%c", star);
        }
      }
   }
```

6. Ce programme fonctionne correctement. Mais il pourrait être amélioré. Tout d'abord, il n'est pas nécessaire d'initialiser la variable `x` à 1. Elle est initialisée à 0 dans la boucle `for`. La déclaration de la variable `tally` en statique est inutile, car le mot clé `static` dans la fonction `main()` n'a aucun effet.

7. Les deux variables `star` et `dash` ne sont pas initialisées.

8. Ce programme affiche les caractères suivants sans jamais s'arrêter (voir exercice 10).

```
X==X==X==X==X==X==X==X==X==X==X==X==X...
```

9. Le problème de ce programme est la portée globale de `ctr`. Les deux fonctions `main()` et `print letter2` utilisent `ctr` en même temps dans une boucle. Etant donné que `print letter2()` change la valeur de `ctr`, la boucle `for` de `main()` n'aura pas de fin. Il y a de nombreuses façons de corriger cette erreur. Une solution consiste à utiliser deux compteurs différents. Une autre solution consiste à changer la portée de `ctr`. On peut la déclarer dans chaque fonction comme variable locale. Voici le code corrigé :

```
#include <stdio.h>
#include <stdlib.h>
void print_letter2(void);      /* Déclaration de la fonction */
int main()
{
   char letter1 = 'X';
   int ctr;
   for(ctr = 0; ctr < 10; ctr++)
   {
     printf("%c", letter1)
     print_letter2();
   }
   exit(EXIT_SUCCESS);
}
void print_letter2(void)
{
   char letter2 = '=';
   int ctr;
   for(ctr = 0; ctr < 2; ctr++)
     printf("%c", letter2);
}
```

Réponses aux questions du Chapitre 13

Quiz

1. Jamais. (Sauf si vous êtes très prudent.)
2. Quand une instruction `break` est exécutée, cela provoque une sortie immédiate de la boucle `for`, `while`, ou `do while` qui contient `break`. Quand une instruction `continue` est exécutée, l'itération suivante de la boucle qui contient `continue` commence immédiatement.
3. Une boucle infinie est une boucle qui ne s'arrête jamais d'elle-même. Elle est créée avec une instruction de boucle `for`, `while`, ou `do while` qui contient un test de condition toujours vrai.
4. La fin d'un programme intervient après la dernière instruction de `main()` ou à l'appel de la fonction `exit()`.
5. L'expression d'une instruction `switch` peut être évaluée avec une valeur de type `long`, `int` ou `char`.
6. L'instruction `default` est une des instructions `case` de `switch`. Cette instruction est exécutée si l'expression de `switch` ne correspond à aucun modèle présenté avec `case`.
7. La fonction `exit()` provoque l'interruption du programme. On peut lui transmettre une valeur qui sera renvoyée au système d'exploitation.
8. La fonction `system()` exécute une commande au niveau du système d'exploitation.

Exercices

1. `continue`;
2. `break`;
3. Sur Unix ou Linux, la réponse serait :

```
system("ls");
```

4. Ces instructions ne contiennent pas d'erreur.
5. L'instruction `default` ne doit pas obligatoirement se trouver à la fin de l'instruction `switch`. Il y a quand même une erreur. Il manque une instruction `break` dans cette instruction `default`.
6. Réponse :

```
if(choix == 1)
   printf("vous avez répondu 1");
```

```
else if (choix == 2)
    printf("vous avez répondu 2");
   else
    printf("vous n'avez choisi ni 1 ni 2");
```

7. Réponse :

```
do {
/* instructions ... */
} while (1);
```

En raison de la multitude de solutions, nous ne présentons pas de réponses pour les questions 9 et 10.

Réponses aux questions du Chapitre 14

Quiz

1. Un flot est une séquence d'octets. Un programme C utilise les flots pour toutes ses entrées/sorties.
2. a) Un clavier est une unité d'entrées.
   b) Un écran est une unité de sorties.
   c) Un disque peut être une unité d'entrées ou une unité de sorties.
3. Tous les compilateurs C supportent les trois flots prédéfinis : `stdin` (le clavier), `stdout` (l'écran), et `stderr` (l'écran).
4. a) `stdout`.
   b) `stdout`.
   c) `stdin`.
   d) N'importe quel flot de sorties : `stdout` ou `stderr`.
5. Les entrées qui passent par la mémoire tampon sont envoyées au programme quand l'utilisateur appuie sur la touche Entrée. Les autres sont envoyées caractère par caractère, dès que la touche est enfoncée.
6. Les entrées qui ont un écho sont recopiées automatiquement dans `stdout` dès que l'utilisateur appuie sur une touche. Les autres sont simplement envoyées dans `stdin`.
7. Vous ne pouvez rendre qu'un caractère entre deux lectures avec la fonction `ungetc()`. Le caractère `EOF` ne peut être rendu avec cette fonction.
8. Avec le caractère de retour à la ligne qui correspond à la touche Entrée.
9. a) Correct.

b) Correct.

c) Correct.

d) q n’existe pas.

e) Correct.

f) Correct.

10. `stderr` ne peut pas être redirigé simplement et est plutôt consacré à l’affichage des erreurs. `stdout` peut être redirigé facilement vers une autre unité de sorties que l’écran avec l’opérateur de redirection >.

Exercices

1. `printf(("Hello, world");`.
2. Réponse :

```
fprintf(stdout, "Hello, world");
puts("Hello, world");
```

3. Réponse :

```
char buffer[31];
scanf("%30[^*]", buffer);
```

4. `printf(("Jack demande, \"qu'est ce qu'un antislash\?\" \ \n Jill répond, \"c'est \'\\\'\"");`.
5. Conseil : utilisez un tableau de 26 caractères. Pour compter chaque caractère, incrémentez l’élément de tableau correspondant à chaque lecture de caractère.
6. Conseil : travaillez avec une chaîne à la fois, puis envoyez une ligne formatée constituée d’un nombre suivi de deux points, puis de la chaîne.

Réponses aux questions du Chapitre 15

Quiz

1.

```
float x;
float *px = &x;
float *ppx = &px;
```

2. Il manque un niveau d’indirection. Il faut écrire :

```
**ppx = 100;
```

3. `tableau` est un tableau de deux éléments dont chacun est lui-même un tableau contenant trois éléments dont chacun contient quatre variables de type `int`.
4. C'est un pointeur vers le premier sous-ensemble de quatre éléments de `tableau`.
5. La première et la troisième.
6. `void fonc(char *p[]);`
7. Telle qu'est écrit son prototype, elle n'a aucun moyen de le savoir.
8. C'est une variable contenant l'adresse du point d'entrée de la fonction.
9. `char (*ptr)(char *x[]);`
10. C'est le prototype d'une fonction renvoyant un pointeur vers un `char`.
11. La structure doit contenir un pointeur vers le même type de structure.
12. Cela signifie que la liste chaînée est vide.
13. Chaque élément de la liste contient un pointeur qui identifie le prochain élément. Le premier élément de cette liste est identifié par le pointeur de tête.
14. a) `var1` est un pointeur vers un entier.
    b) `var2` est un entier.
    c) `var3` est un pointeur vers un pointeur vers un entier.
15. a) `a` est un tableau de 36 (3∞12) entiers.
    b) `b` est un pointeur vers un tableau de 12 entiers.
    c) `c` est un tableau de 12 pointeurs vers des entiers.
16. a) `z` est un tableau de 10 pointeurs vers des caractères.
    b) `y` est une fonction qui accepte un entier (`champ`) comme argument et renvoie un pointeur vers un caractère.
    c) `x` est un pointeur vers une fonction qui accepte un entier (`champ`) comme argument et renvoie un caractère.

Exercices

1. `float (*fonc)(int champ);`.
2. `int (*options de menu[10])(char *titre);`.

   Le nom choisi implique l'utilisation possible de cette déclaration : le numéro du menu sélectionné pourrait se rapporter à l'index dans le tableau pour le pointeur vers la fonction. Ainsi, on pourrait exécuter la fonction pointée par le cinquième élément du tableau si c'était le menu numéro 5 qui avait été sélectionné par l'utilisateur.
3. `char *ptrs[10];`.

4. ptr n'a pas été déclaré comme un pointeur vers un tableau de 12 entiers. Le code correct serait donc :

```
int x[3][12];
int (*ptr)[12];

ptr = x;
```

5. Voici l'une des nombreuses solutions possibles :

```
struct friend {
   char name[35+1];
   char street1[30+1];
   char street2[30+1];
   char city[15+1];
   char state[2+1];
   char zipcode[9+1];
   struct friend *next;
```

Réponses aux questions du Chapitre 16

Quiz

1. Un flot en mode texte effectue automatiquement la traduction nécessaire du caractère \n (newline) que C utilise pour marquer la fin d'une ligne et le groupe <retour chariot><à la ligne> que les systèmes issus de DOS comme Windows emploient pour le même résultat. À l'opposé, un flot en mode binaire n'effectue aucune traduction. Tous les octets sont écrits ou lus sans aucune transformation. Sur Linux, texte et binaire revient au même : binaire.
2. Ouvrir le fichier à l'aide d'un appel à la fonction `fopen()`.
3. Pour appeler `fopen()`, vous devez spécifier le nom du fichier disque sur lequel vous voulez travailler et le mode dans lequel vous voulez l'ouvrir.
4. Accès formaté, accès par caractère et accès direct.
5. Accès séquentiel et accès aléatoire.
6. `EOF` est une constante symbolique qui représente la marque de fin de fichier et vaut symboliquement –1.
7. `EOF` est utilisé avec des fichiers texte pour savoir si on a atteint la fin du fichier.
8. En mode binaire, en appelant la fonction `feof()`. En mode texte, on peut l'utiliser aussi ou faire une comparaison du caractère lu avec `EOF`.

9. L’indicateur de position du fichier indique la position dans un fichier où aura lieu la prochaine lecture ou écriture. Sa valeur peut être modifiée par un appel à `rewind()` ou `fseek()`.
10. À l’ouverture, l’indicateur de position pointe sur le premier caractère du fichier et vaut donc `SEEK SET` (c’est-à-dire 0) sauf si vous ouvrez un fichier existant en mode Append (mise à jour). Dans ce cas, l’indicateur de position vaut `SEEK CUR` (c’est-à-dire position courante).

Exercices

1. `rewind(p);` ou `fseek(p, 0, SEEK SET)`;.
2. Vous ne pouvez pas utiliser un test par `EOF` sur un fichier binaire.

Réponses aux questions du Chapitre 17

Quiz

1. C’est le nombre de caractères compris entre le début et la fin de la chaîne plus 1 pour tenir compte du zéro terminal. Elle se détermine par l’opérateur `sizeof()`.
2. Il faut être sûr qu’il y a assez de place.
3. Mettre bout à bout deux chaînes (ne figure pas au NPLI).
4. Le code ASCII d’un des caractères de la chaîne est supérieur à celui du caractère occupant la même position dans l’autre chaîne.
5. `strcmp()` compare la totalité de deux chaînes alors que `strncmp()` ne compare que le nombre de caractères spécifié des deux chaînes.
6. `strcmp()` compare deux chaînes en faisant une distinction entre les majuscules et les minuscules, alors que, pour `strcmpi()`, il n’y a pas de différence (`'A'` et `'a'` sont identiques).
7. `isascii()` regarde si le code ASCII du caractère est compris entre 0 et 127.
8. `isascii()` et `iscntrl()`.
9. `isalnum()`, `isalpha()`, `isascii()`, `isgraph()`, `isprint()` et `isupper()`. (65 est le code ASCII de ’A’).
10. À reconnaître si un caractère donné remplit une certaine condition telle qu’appartenir à l’ensemble des lettres, être un signe de ponctuation, etc.

Exercices

1. vrai et faux (respectivement non zéro et zéro).
2. Voici les réponses :
   a) 65.
   b) 81.
   c) –34.
   d) 0. (`errno` sera non nul)
   e) 12.
   f) 0. (`errno` sera non nul)
3. Voici les réponses :
   a) 65.000000.
   b) 81.230000.
   c) –34.200000.
   d) 0.000000. (`errno` sera non nul)
   e) 12.000000.
   f) 1000.000000.
4. Aucune place n'a été allouée à `string2` avant de l'utiliser. Il n'existe aucun moyen de savoir où `strcpy()` copie la valeur de `string1`. Pire, vous risquez une faute de segmentation (écriture à un endroit interdit).

Réponses aux questions du Chapitre 18

Quiz

1. Passer par valeur signifie qu'on va effectuer une copie des valeurs des arguments. Passer par référence signifie que la fonction va recevoir les adresses des arguments. La seconde méthode permet à la fonction de modifier la valeur des arguments, chose impossible avec la première méthode.
2. C'est un pointeur qui peut pointer sur n'importe quel type d'objet C (un pointeur générique).
3. L'usage le plus courant est dans la déclaration des arguments d'une fonction, ce qui permet à la fonction de manipuler divers types d'objets C.

4. Un pointeur de type `void` pouvant pointer sur n'importe quel type d'objet C, il faut le "personnaliser" lorsqu'on veut l'appliquer à un objet C particulier (ne serait-ce qu'en raison de la taille de l'objet en question).
5. Non ; elle doit avoir au minimum un argument "constant" (fixe) pour l'informer du nombre d'arguments qui lui seront passés à chaque appel.
6. Il faut d'abord appeler `va start()` pour initialiser la liste d'arguments. Ensuite, on appellera `va arg()` pour récupérer les arguments et, finalement, `va end()` pour faire le nettoyage final.
7. Question piège ! Un pointeur de type `void` qui n'a pas été casté à un type de données C défini ne peut pas être incrémenté.
8. Pourquoi pas ? C'est un type C comme un autre.

Exercices

1. `int fonc(char tableau[]);`

   (Remarquez que `int` est superflu puisque, par défaut, la valeur de retour d'une fonction est de type `int`.)
2. `int nombres(int *nb1, int *nb2, int *nb3);`
3. `nombres(&ent1, &ent2, &ent3);`
4. Si bizarre que cela puisse paraître, ces deux lignes ne contiennent aucune erreur. Le premier et le troisième astérisque sont des opérateurs d'indirection ; le deuxième est l'opérateur "produit" associé au signe d'affectation. Comme son nom l'indique, la fonction effectue bien une élévation au carré de l'argument qui lui est passé par adresse.
5. Il manque un appel initial à `va start()` et un appel final à `va end()`.

Réponses aux questions du Chapitre 19

Quiz

1. `double`.
2. Avec la plupart des compilateurs, `time t` est équivalent à `long`. Mais c'est loin d'être garanti ! Vous pouvez consulter la fichier d'en-tête time.h fourni avec votre compilateur ou son manuel de référence pour avoir un renseignement plus précis.
3. `time()` renvoie le nombre de secondes écoulées depuis le 1er janvier 1970. `times()` renvoie un nombre de tops d'horloge correspondant au temps alloué à votre

programme, `clock()` renvoie le nombre (approximatif) de ticks d'horloge écoulées depuis le début de l'exécution du programme.

4. Rien du tout ; elle ne fait qu'informer.
5. Trier le tableau en ordre ascendant.
6. 14.
7. 4.
8. 21.
9. 0 si les valeurs comparées sont égales; un nombre négatif, si le premier élément est inférieur au second ; un nombre positif, si le premier élément est supérieur au second.
10. `NULL`.

Exercices

1. Voici ce qu'il faut écrire :

```
bsearch(monnom, names, (sizeof(names)/sizeof(names[0])),
        sizeof(names[0]), comp_names);
```

2. Il y a trois problèmes. D'abord, on a oublié d'indiquer la taille du tableau dans l'appel à `qsort()`. Ensuite, toujours dans l'appel à `qsort()`, il ne faut pas taper de parenthèses à la suite du nom de la fonction de comparaison. Enfin, la fonction de comparaison manque dans le programme : `compare function()` ne se trouve pas dans le programme.
3. Cette fonction de comparaison renvoie de mauvaises valeurs et il faut croiser les valeurs renvoyées au cas où le résultat de la comparaison ne serait pas égalité.

Réponses aux questions du Chapitre 20

Quiz

1. `malloc()` alloue un bloc de mémoire de la taille indiquée en argument et le laisse tel quel alors que les arguments passés à `calloc()` indiquent le nombre d'éléments d'une certaine taille à allouer. En outre, `calloc()` remet le bloc alloué à zéro.
2. La précision des calculs, particulièrement dans le cas d'une division, lorsqu'il s'agit de variables de type `int` qui sont alors *castées* en `float` ou en `double`.
3. a) `long`.

   b) `int`.

c) `char`.

d) `float`.

e) `float`.

4. C'est de la mémoire allouée au moment de l'exécution du programme. Ce procédé permet d'allouer juste la quantité de mémoire nécessaire.

5. `memmove()` donne un résultat correct même si source et destination se recouvrent partiellement, ce qui n'est pas le cas de `memcpy()`. Lorsqu'il n'y a pas recouvrement, les deux fonctions sont identiques.

6. En définissant un champ de bits de 3 octets. 2 puissance 3 étant égal à 8, ce champ pourra stocker des valeurs entre 1 et 7.

7. 2 octets. En utilisant les champs de bits, vous pouvez déclarer la structure suivante :

```
struct date{
unsigned month : 4;
unsigned day   : 5;
unsigned year  : 7;
}
```

Cette structure enregistre la date sur 2 octets (16 bits). Le champ `month` de 4 bits peut recevoir des valeurs entre 0 et 15, ce qui permet d'enregistrer les douze mois de l'année. De la même façon, le champ `day` de 5 bits peut recevoir des valeurs entre 0 et 31 et le champ `year` de 7 bits peut recevoir les valeurs de 0 à 127. Nous supposons que l'année sera ajoutée à la valeur 1900 pour représenter les années 1900 à 2027.

8. 00100000

9. 00001001

10. Ces deux expressions ont la même valeur. Dans les deux cas, tous les bits de la valeur initiale sont inversés.

Exercices

1.

```
long *ptr;
ptr = malloc(1000 * sizeof(long);
```

2.

```
long *ptr;
ptr = calloc(1000, sizeof(long));
```

3. Première solution, avec une boucle `for` :

```
int i;
for (i=0; i<1000; i++)
 data[i] = 0;
```

Seconde solution, plus rapide, en utilisant la fonction `memset()` :

```
memset(data, 0, 1000 * sizeof(float));
```

4. Ce court programme ne contient pas d'erreurs de syntaxe, mais le résultat obtenu sera peu précis puisque le quotient de deux entiers est un entier. La partie fractionnaire sera perdue. Il aurait fallu écrire :

```
reponse = (float) nombre1 / nombre2;
```

5. `p` étant du type `void`, il doit être casté avant d'être utilisé, et il faudrait écrire :

```
*(float *)p = 1.23;
```

6. Non, vous devez d'abord placer les champs de bit dans une structure. Voici le code corrigé :

```
struct quiz_answers {
unsigned answer1   : 1;
unsigned answer2   : 1;
unsigned answer3   : 1;
unsigned answer4   : 1;
unsigned answer5   : 1;
char student_name[15];
```

Réponses aux questions du Chapitre 21

Quiz

1. La programmation modulaire consiste à répartir les instructions d'un programme en plusieurs modules (fichiers sources).
2. Le module principal est celui qui contient la fonction `main()`.
3. Pour éviter les méfaits d'un éventuel effet de bord. Ainsi, les arguments passés à la macro seront évalués en premier.
4. Une macro augmente l'encombrement du programme, mais raccourcit son temps d'exécution.
5. `defined` regarde si le nom qui le suit a été "défini" (par un `#define`). Il renvoie `TRUE` si c'est le cas ; `FALSE`, sinon.
6. `#end`.
7. Ils ont l'extension .o (ou .obj avec certains compilateurs).

8. `#include` recopie le contenu du fichier spécifié dans le programme source où se trouve cette directive.
9. Lorsque le nom de fichier est encadré par < et >, le fichier est recherché dans le répertoire standard des fichiers d'include. Lorsqu'il est encadré par des guillemets, il est d'abord recherché dans le répertoire courant.
10. Cette macro insère la date de compilation du programme à l'endroit où elle apparaît.

Index

Symboles

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

U

V

W

Le Programmeur

Le langage C

Cet ouvrage fondamentalement pratique est une excellente introduction pour tous ceux qui souhaitent s'initier rapidement et efficacement à la programmation en C.

Grâce à des exercices pratiques et des cas concrets, il vous initie progressivement à toutes les bases du langage (fonctions, structures, pointeurs, gestion mémoire, gestion fichiers, bibliothèques de classes, etc.), vous apprend à utiliser les bonnes syntaxes et vous fournit de nombreux conseils, notamment en matière de sécurité.

À l'issue de cet ouvrage, vous serez apte à réaliser de petits programmes et à comprendre le code des plus gros. À l'aide de bibliothèques de fonctions existantes, vous pourrez également créer vos interfaces graphiques, communiquer avec d'autres programmes sur Internet, réaliser des jeux ou traiter des données issues des bases de données.

Cette nouvelle édition bénéficie d'une révision complète du code en faveur d'une syntaxe plus précise et plus concise. Les exemples et explications ont également été actualisés.

- **Impression des listings**
- **Structure d'un programme C**
- **Constantes et variables numériques**
- **Instructions, expressions et opérateurs**
- **Le nombre mystère**
- **Les fonctions**
- **Les instructions de contrôle**
- **Les principes de base des entrées/sorties**
- **Utilisation des tableaux numériques**
- **Les pointeurs**
- **Une pause**
- **Caractères et chaînes**
- **Les structures**
- **La portée des variables**
- **Les messages secrets**
- **Les instructions de contrôle (suite)**
- **Travailler avec l'écran, l'imprimante et le clavier**
- **Retour sur les pointeurs**
- **Utilisation de fichiers sur disque**
- **Comptage des caractères**
- **Manipulation de chaînes de caractères**
- **Retour sur les fonctions**
- **Exploration de la bibliothèque des fonctions**
- **Calcul des versements d'un prêt**
- **La mémoire**
- **Utilisation avancée du compilateur**
- **Charte des caractères ASCII**
- **Mots réservés**
- **Travailler avec les nombres binaires et hexadécimaux**
- **Portabilité du langage**
- **Fonctions C courantes**
- **Bibliothèques de fonctions**
- **Les logiciels libres**

Niveau : Débutant / Intermédiaire

Catégorie : Programmation
Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4085-6

